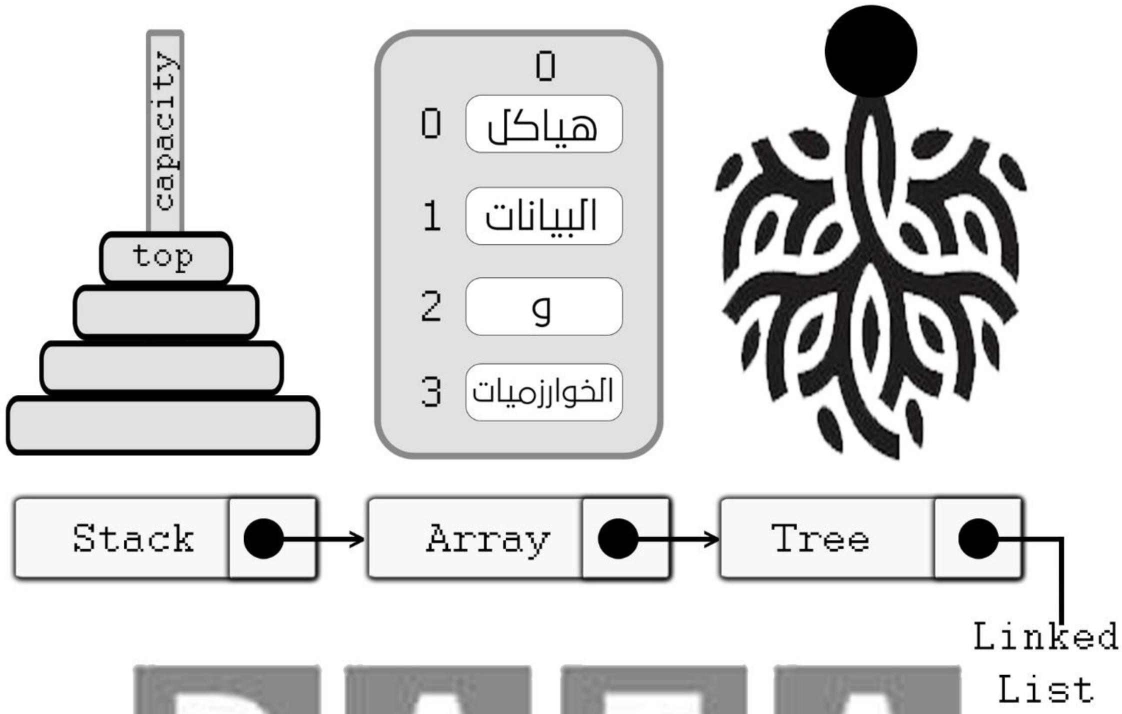


DATA STRUCTURE

هياكل البيانات والخوارزميات

Data Structures and Algorithms

Java

للدكتور: عبد الناصر الغويل

تعديل الطالب: علي العمامي

مراجعة للمصفوفات بلغة الجافا Arrays

نظرا لأهمية موضوع المصفوفات كهيكل من هياكل البيانات سنقوم أولا بمراجعتها بشكل مفصل ليتسنى لنا كتابة البرامج المختلفة بشكل أسهل حيث أننا سنقوم باستخدامها بشكل كبير في هذا الفصل، كما سنقوم أيضا بمراجعة الاستثناءات.

عادة ما تستخدم برامج الجافا أسماء متعددة لتعريف الحقول والمتغيرات، ولكن ماذا لو أردنا أن نستخدم في إحدى البرامج 500حقل مثلا أو أكثر، لغة الجافا مثل أغلب اللغات توفر لنا ما يسمى بالمصفوفات لكي نستخدم اسم واحد لمجموعة من المتغيرات ويتم التمييز بينها باستخدام رقم الفهرس "index" للوصول إلى العنصر المطلوب. سنقوم بمراجعة نوعين من المصفوفات وهي: المصفوفات الأحادية والمصفوفات الثنائية، كما تجدر الإشارة إلى أنه يمكن تكوين مصفوفات ذات أبعاد أكثر بنفس الطريقة.

المصفوفات الأحادية One Dimensional Array

هي مجموعة من العناصر من نفس النوع يتم تخزينها في صف أو عمود ذات طول ثابت لها اسم واحد، ويتم الوصول إلى أي عنصر باستخدام اسم المصفوفة وترتيب العنصر داخل المصفوفة أو ما يسمى بفهرس أو دليل المصفوفة العناصر داخل المصفوفة قد تكون من النوع الأساسي "Primitive Type" أو مرجع إلى كائن "Object".

الشكل العام لتعريف المصفوفة الأحادية:

```
Type[] arrayName;
```

حيث:

`Type`: نوع بيانات العناصر التي سيتم تخزينها بالمصفوفة، مثلا: `int` أو `double` أو `boolean` ...إلخ.

`arrayName`: هو اسم المصفوفة، وهو مرجع لكائن، حتى وإن كانت العناصر المخزنة به من النوع الأساسي، أو يمكن تعريفها كالتالي:

```
Type arrayName[];
```

ولكن لا شيء موجود بها حتى الآن إلى أن يتم بناء المصفوفة باستخدام `new` كالتالي:

```
Type[] arrayName = new Type[integer-expression];
```

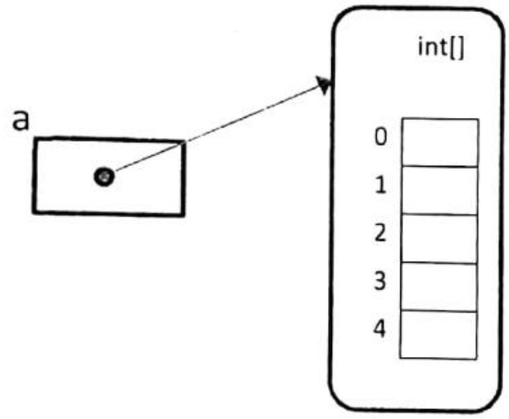
فمثلا لتعريف مصفوفة اسمها a تتكون من خمس أعداد صحيحة يمكننا كتابتها كالتالي:

```
int[] a = new int[5];
```

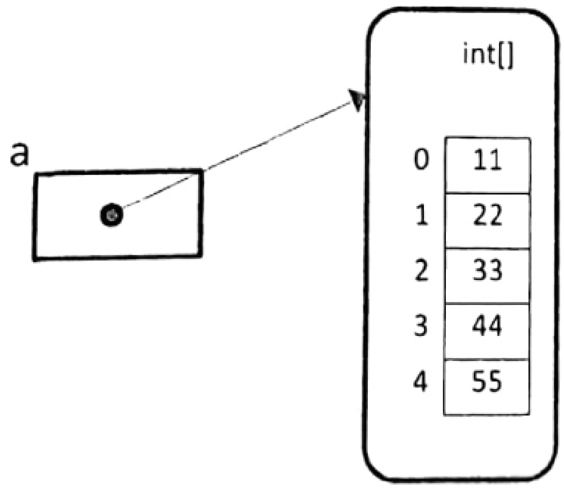
أو كالتالي:

```
int[] a;
a = new int[5];
```

هنا سيتم تكوين مصفوفة أحادية تحتوي على خمس عناصر صحيحة وسيتم تمهيد العناصر وإعطاءها قيمة 0 تلقائيا. أما إذا كانت العناصر عشرية فسيتم إعطاءها 0.0، وإذا كانت العناصر منطقية فسيتم تعبئتها بـ false. أما إذا كانت مصفوفة من الكائنات فسيتم تعبئتها بـ null تلقائيا عند تكوينها. فتعريف المصفوفة السابق يكون كالتالي:



لاحظ أن فهرس المصفوفة في الجافا مثل لغة السي تبدأ من صفر وينتهي بطول المصفوفة -1، فإذا تمت تعبئة المصفوفة بالقيم الصحيحة سيكون شكلها كالتالي:



فللوصول إلى العنصر الثالث (العنصر ذو الفهرس 2) و إسناده إلى المتغير الصحيح k نكتب:

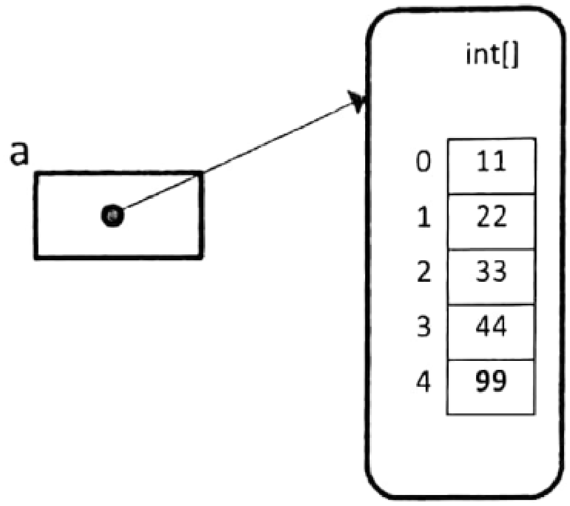
```
int k = a [2];
```

المتغير k الآن سيحتوي على القيمة 33.

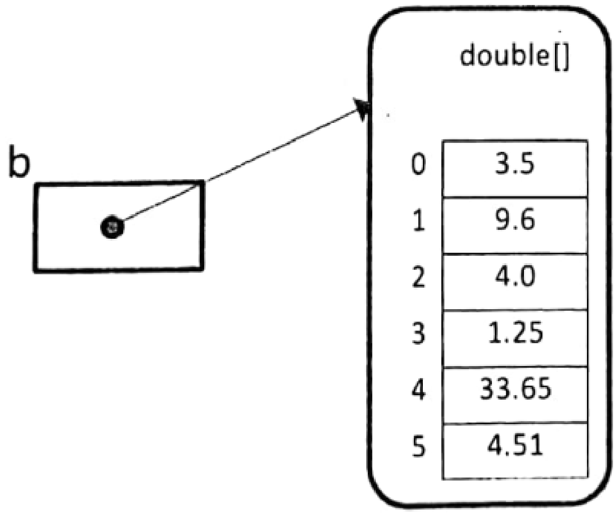
أما لو أردنا تخصيص قيمة لعنصر من عناصر المصفوفة فنكتب:

```
a[4] = 99;
```

يعني خزن القيمة 99 في موقع المصفوفة ذو الفهرس 4، فتصبح كالتالي:

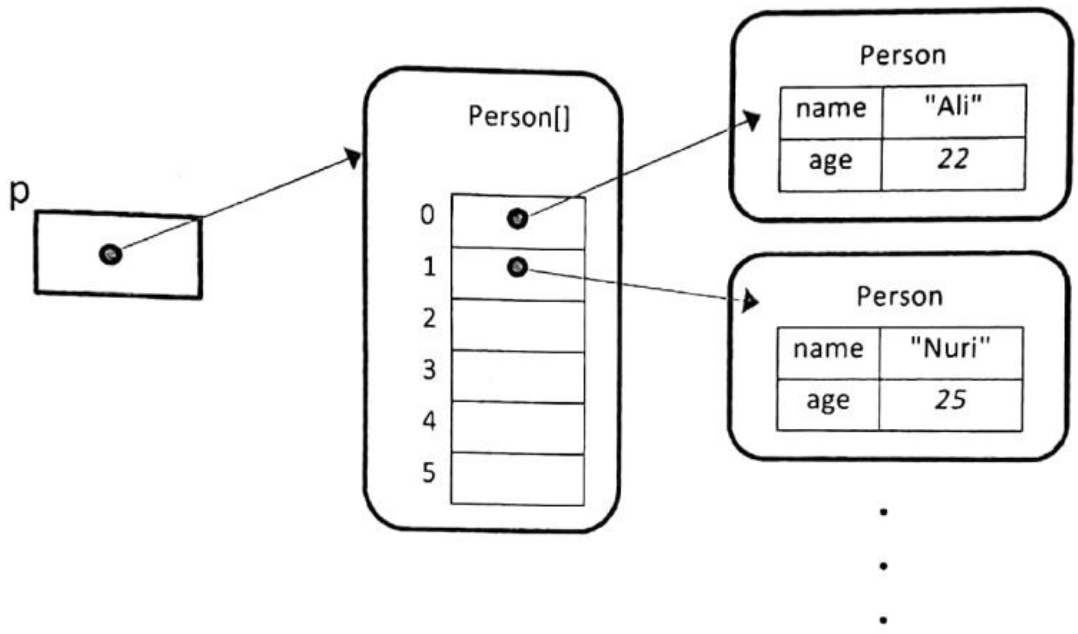


ولتعريف مصفوفة من النوع العشري بها 6 عناصر يكون كالتالي:



أما لتعريف مصفوفة من الكائنات (لنفترض أن لدينا فصيل الشخص "Person" له حقلين هما الاسم "name" و العمر "age" ونريد تكوين مصفوفة من كائنات الشخص) فيتم كالتالي:

```
Person[] p = new Person[6];
```



فللوصول لاسم الشخص الثاني (العنصر ذو الفهرس 1) و إسناده لمتغير من نوع السلسلة النصية name نكتب:

```
String name = p[1].getName();
```

هنا المتغير name سيخصص له القيمة "Nuri".

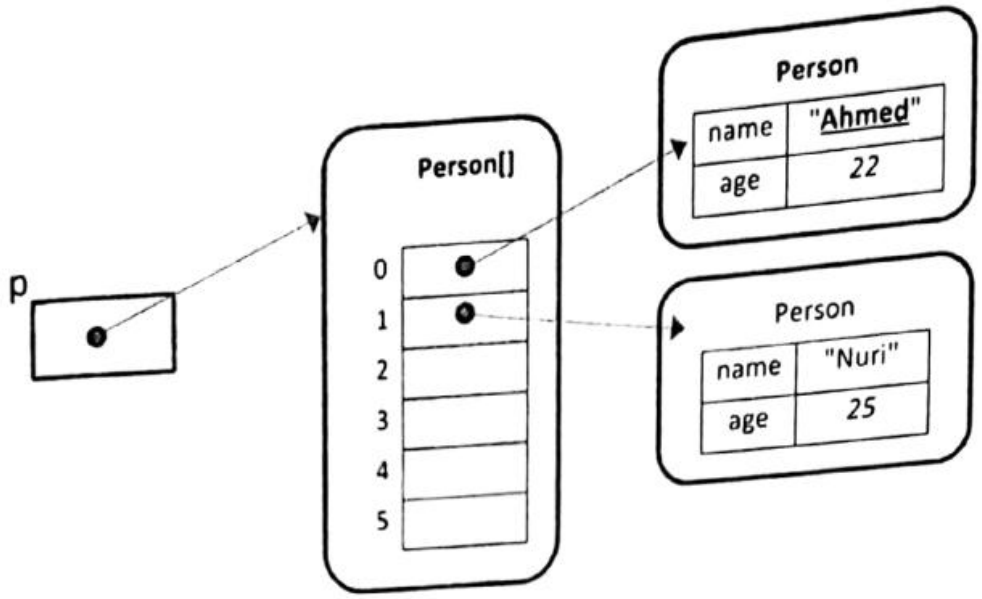
وللوصول إلى عمر الشخص الأول (ذو الفهرس 0) وإسناده للمتغير ag:

```
int ag = p[0].getAge( );
```

ولتخزين الاسم "Ahmed" في الشخص الأول بالمصفوفة (ذو الفهرس 0) فنكتب:

```
po[0].setName( "Ahmed");
```

لاحظ أننا استخدمنا طرق الـ get و الـ set لـ لوصول/تغيير قيم الحقول؛ لأن الحقول عادة ما تكون خاصة "private" وهذه الطرق تكون عامة "public" وتستخدم لتغيير أو الوصول إلى الحقول وبالتالي ستكون النتيجة هي:



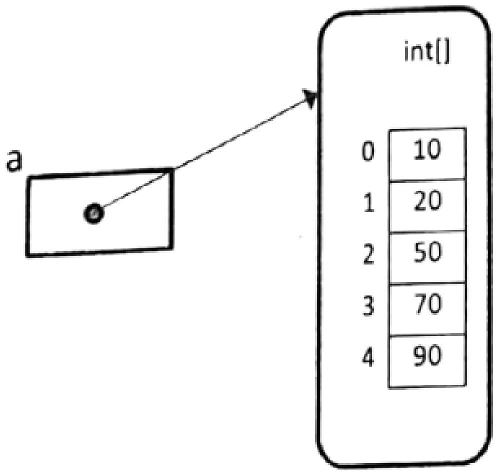
ملاحظة:

كل المصفوفات لها حقل صحيح عام اسمه `length` يقوم بترجيع طول المصفوفة.

كما يمكن تعريف المصفوفات بطريقة أخرى وهي إعطائها قيم مبدئية عند تعريفها كالتالي:

```
int[] a = {10, 20, 50, 70, 90};
```

هنا قمنا بتعريف وتخصيص قيم مبدئية للمصفوفة في نفس الوقت، المترجم سوف يحسب عدد عناصر المصفوفة تلقائيا ومن ثم يحدد حجم المصفوفة، وفي حالتنا هذه 5 عناصر كالتالي:



لاحظ أننا نقوم بفصل قيم عناصر المصفوفة عن بعضها بالفاصلة (,) وتكون القيم محصورة بين أقواس الفئة {}.

كما يمكن مثلا تعريف مصفوفة من السلاسل النصية وتمهيدها عند التعريف كالتالي:

```
String[] monthNames= { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
```

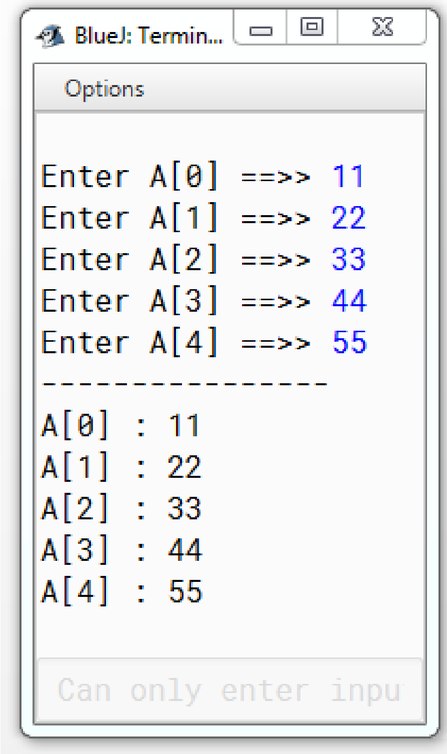
مثال:

اكتب برنامج جافا مكون من فصيل اسمه Array1 يحتوي فقط على الدالة الرئيسية main() لإدخال خمسة عناصر صحيحة في مصفوفة أحادية وطباعتها.

```
import java.util.Scanner;
public class Array1
{
    public static void main (String[] args)
    {
        int [] a = new int[5];
        Scanner kb = new Scanner (System.in);
        System.out.println ('\f');

        // القراءة أو الإدخال في مصفوفة
        for (int i = 0; i< a.length ; i++)
        {
            System.out.print("Enter A[" + i + "] ==>> ");
            a[i] = kb.nextInt();
        }
        System.out.println("----------------");

        // طباعة عناصر مصفوفة
            for (int i = 0; i < a.length; i++)
                System.out.println ("A[" + i + "] : " + a[i]);
    }
}
```



لاحظ أننا استخدمنا a.length والذي يقوم بإرجاع طول المصفوفة، أيضا استخدمنا حلقة for للمرور على عناصر المصفوفة واحد تلو الأخر.

بدل من استخدام العدد 5 لتحديد طول المصفوفة، يمكن استخدام ثابت لتحديد طول المصفوفة كالتالي:

```
final int MAX = 5;
int[] a = new int [MAX];
```

كما يمكننا أيضا (من جافا 5 وبما بعدها من الإصدارات) استخدام حلقة for المحسنة "Enhanced for"، بنيتها كالتالي:

```
for ( Type name : arrayName)
{
    عدة جمل;
}
```

حيث أن:

Type: هو نوع البيانات والذي يأخذ نفس نوع بيانات المصفوفة.

name: اسم المتغير الذي سيحتوي على بيانات عناصر المصفوفة.

arrayName: اسم المصفوفة المراد المرور على عناصرها عنصر تلو الأخر.

فمثلا إذا كانت لدينا حلقة for التالية لطباعة عناصر المصفوفة:

```
for (int i = 0 ; i < a.length ; i++)
{
    System.out.println(a[i]);
}
```

يمكننا كتابتها بحلقة for المحسنة كالتالي:

نفس نوع بيانات المصفوفة

متغير سيحتوي على قيمة عنصر المصفوفة

اسم المصفوفة

طباعة *data* قيمة عنصر المصفوفة

```
for (int data : a)
{
    System.out.println(data);
}
```

هنا سيتم المرور على عناصر المصفوفة عنصر تلو الأخر وفي كل مرة سيحتوي المتغير data على قيمة عنصر المصفوفة، فمثلا: في الحلقة الأولى سيحتوي على a[0]، ثم في الحلقة الثانية سيحتوي على a[1] ... وهكذا

يمكننا إعادة المثال السابق كالتالي:

```
import java.util.Scanner;
public class Array11
{
    public static void main (String[] args)
    {
        final int MAX = 5;
        int [] a = new int[MAX];

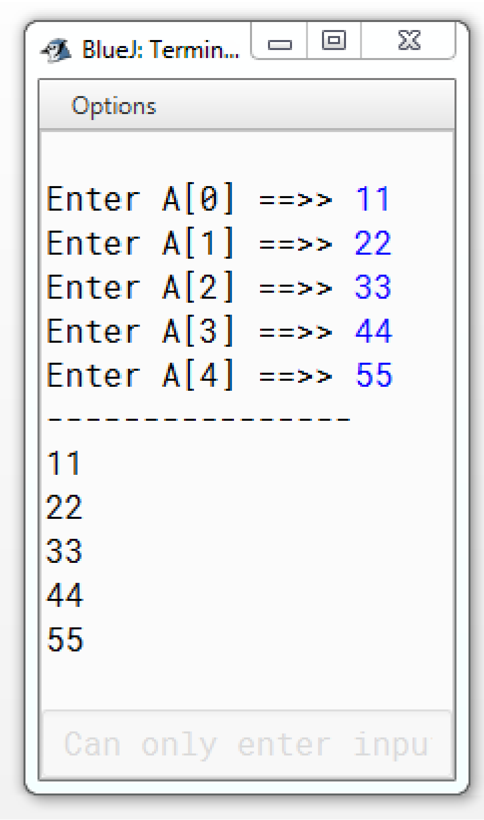
        Scanner kb = new Scanner (System.in);
```

```
        System.out.println ('\f');

        // القراءة أو الإدخال في مصفوفة
        for (int i = 0; i< a.length ; i++)
        {
            System.out.print("Enter A[" + i + "] ==>> ");
            a[i] = kb.nextInt();
        }

        System.out.println("----------------");

            //المحسنة for طباعة عناصر مصفوفة باستخدام
            for (int data : a)
                System.out.println (data);
    }
}
```



أما لطباعة الفهرس مع القيمة في جملة for المحسنة فيمكننا استخدامها كالتالي:

```
int i = 0;
for (int data : a)
{
    System.out.println ("A[" + (i++) + "] : " + data);
}
```

مثال:

اكتب برنامج جافا مكون من فصيل اسمه *Array2* يحتوي فقط على الدالة الرئيسية *main*() لإدخال خمسة عناصر من نوع السلسلة النصية في مصفوفة أحادية ومن ثم إدخال سلسلة أخرى والبحث عنها ضمن عناصر المصفوفة وطباعة ما إذا كانت موجودة أم لا.

```
import java.util.Scanner;
public class Array2
{
    public static void main (String[] args)
    {
        String[] a = new String[5];
        System.out.println ('\f');
        Scanner kb = new Scanner (System.in);

        // القراءة أو الإدخال في مصفوفة
        for (int i = 0; i< a.length ; i++)
        {
            System.out.print("Enter String[" + i + "] ==>> ");
            a[i] = kb.nextLine();
        }
```

```
        // السلسلة المراد البحث عنها
        System.out.print("\n Enter a new String : ");
        String lookFor = kb.nextLine();

            boolean isFound = false;

            // حلقة للبحث عن السلسلة ضمن المصفوفة وعند وجودها يتم الخروج من الحلقة مباشرة
            for (String data : a)
            {
                if (lookFor.equals(data))
                {
                    isFound = true;
                    break;
                }
            }

        // طباعة ما إذا كانت السلسلة موجودة أم لا
        if (isFound)
            System.out.println ("The item is found ...");
        else
            System.out.println ("The item is Not found ...");
    }
}
```

مثال:

اكتب برنامج جافا مكون من فصيل اسمه Array3 يحتوي فقط على الدالة الرئيسية *main()* لإدخال ستة عناصر حقيقية (أرقام صحيحة) تمثل أسعار سلع، في مصفوفة أحادية وإيجاد مجموعها وطباعة الأسعار عكسيا.

```
import java.util.Scanner;
public class Array3
{
    public static void main (String[] args)
    {
        final int MAX = 6;
        double[] price = new double[MAX];
        Scanner kb = new Scanner (System.in);
        System.out.println ('\f');

        // القراءة أو الإدخال في مصفوفة
        for (int i = 0; i< price.length ; i++)
        {
            System.out.print("Enter String[" + i + "] ==>> ");
            price[i] = kb.nextDouble();
        }

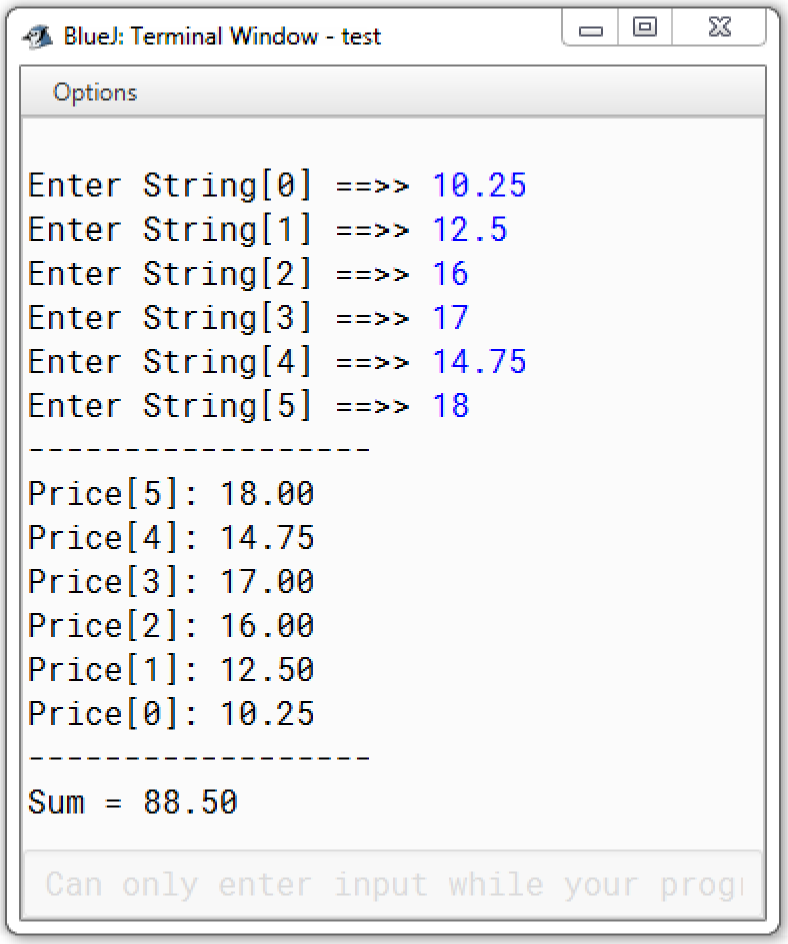
        // المعالجة/إيجاد مجموعة الأسعار
        double sum = 0.0;
        for (int i = 0 ; i< price.length ; i++)
            sum += price[i];

        System.out.print("------------------");

            //الطباعة العكسية
```

```
        for (int i = price.length-1 ; i>=0 ; i--)
        {
           System.out.printf("Price[%d]: %.2f\n" ,i, price[i]);
        }

        // طباعة المجموع
        System.out.println("------------------");
        System.out.printf("Sum = %5.2f" , sum);
    }
}
```



لاحظ أن برامج المصفوفات عادة تتكون من ثلاثة أجزاء رئيسية هي الإدخال والمعالجة والطباعة.

مثال:

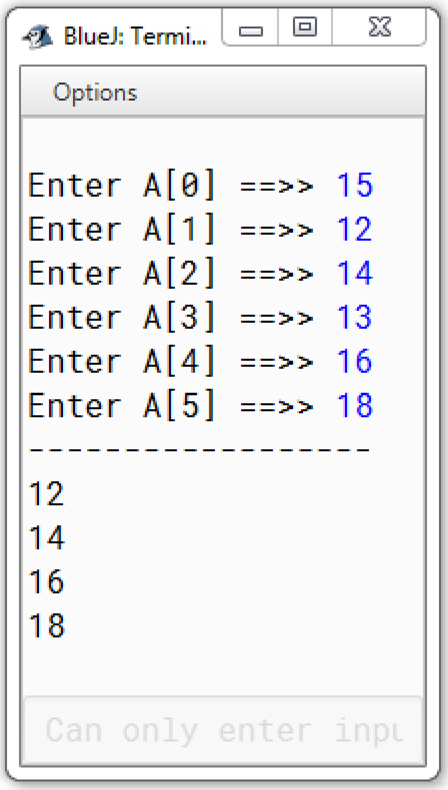
أكتب برنامج جافا مكون من فصيل اسمه Array4 يحتوي على الدالة الرئيسية *main*() لإدخال ستة عناصر صحيحة في مصفوفة أحادية وطباعة العناصر الزوجية فقط.

```
import java.util.Scanner;
public class Array4
{
    public static void main (String[] args)
    {
        final int MAX = 6;
        int[] a = new int[MAX];

        Scanner kb = new Scanner (System.in);
        System.out.println ('\f');

        for (int i = 0; i < a.length; i++)
        {
            System.out.print("Enter A[" + i + "] ==>> ");
            a[i] = kb.nextInt();
        }

        System.out.println("------------------");
        for (int data : a)
        {
            if (data % 2 == 0)  //إذا قبل العنصر القسمة على 2 بدون باقي فهذا يعني أنه عدد زوجي
                System.out.println (data);
        }
    }
}
```



مثال:

اكتب برنامج جافا مكون من فصيلين الأول اسمه الشخص "***Person***" يحتوي على حقلي رقم ***no*** واسم ***name*** الشخص، كما يحتوي على باني واحد يستقبل قيمتين مناظرتين للحقلين، أيضا يحتوي على طرق "Methods" الـ get والـ set لكلا الحقلين. أما الفصيل الثاني فاسمه: ArrayOfPersons يقوم بتكوين مصفوفة من كائنات الأشخاص حجمها ثلاث أشخاص، ويحتوي على الباني الذي يتم فيه قراءة قيم تمثل أرقام وأسماء الأشخاص الثلاثة وتخزينها بالمصفوفة السابقة الذكر، ومن ثم طباعة البيانات المدخلة.

الفصيل الأول Person:

```
public class Person
{
    private int no;
    private String name;

    public Person (int n1, String n2)
```

```
    {
        no   = n1;
        name = n2;
    }

    public String getName()
    {
        return name;
    }

    public int getNo()
    {
        return no;
    }

    public void setName(String n)
    {
        name = n;
    }

    public void setNo(int n)
    {
        no = n;
    }
}
```

الفصيل الثاني ArrayOfPersons:

```
import java.util.Scanner;
public class ArrayOfPersons
{
    final int MAX = 3;

    //تعريف مصفوفة الأشخاص خارج الباني
    Person[] people = new Person[MAX];
    Scanner kb = new Scanner (System.in);

    public ArrayOfPersons()
        {
            System.out.print('\f');

             //تصفير عناصر المصفوفة
            for (int i = 0 ; i < MAX ; i++)
            {
                people[i] = null;
            }

            for(int i = 0; i < MAX ; i++)
            {
                //إدخال بيانات الشخص من الشاشة
           System.out.println("Enter Person [" + (i+1) + "]:");
                System.out.println("\t\t Enter Number ===>>" );
                int no = kb.nextInt();
                kb.nextLine();
                System.out.print("\t\t Enter Name ===>> " );
```

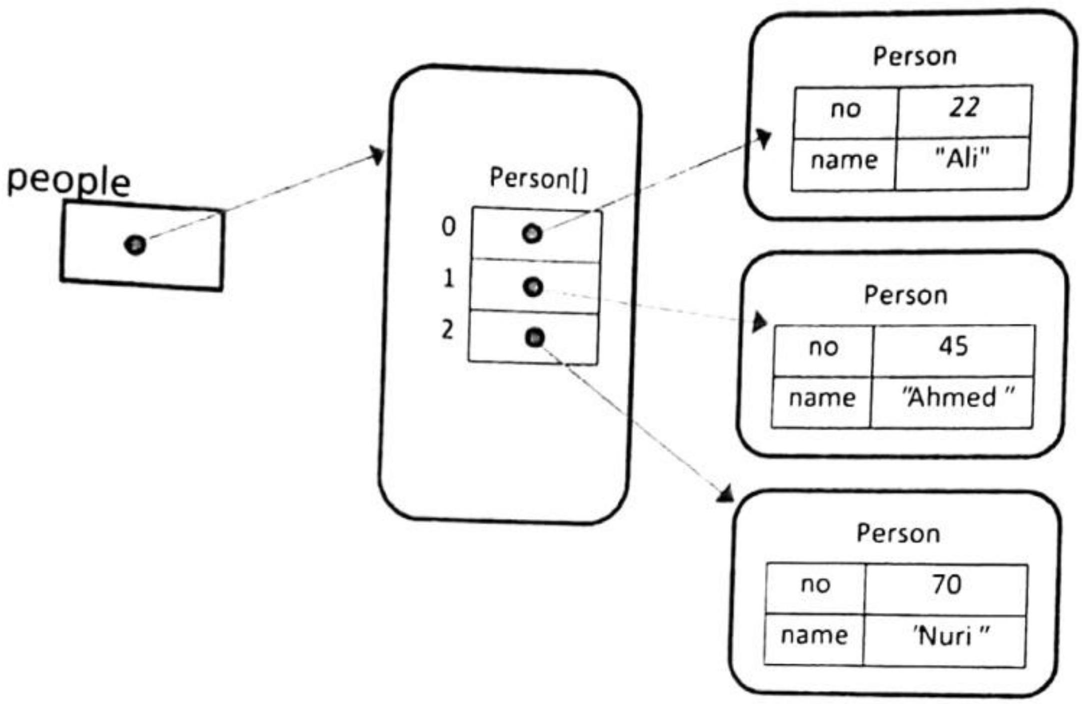
```
            String name = kb.nextLine();
            //تكوين كائن شخص جديد من البيانات المدخلة
            Person p  = new Person (no, name);
            // جعل عنصر المصفوفة يشير إلى الكائن الجديد
            people[i] = p;
        }

        for (int i = 0; i < MAX ; i++)
        {
            System.out.format("Number = %d \t Name = %s \n"
, people[i].getNo() , people[i].getName());
        }
    }
}
```

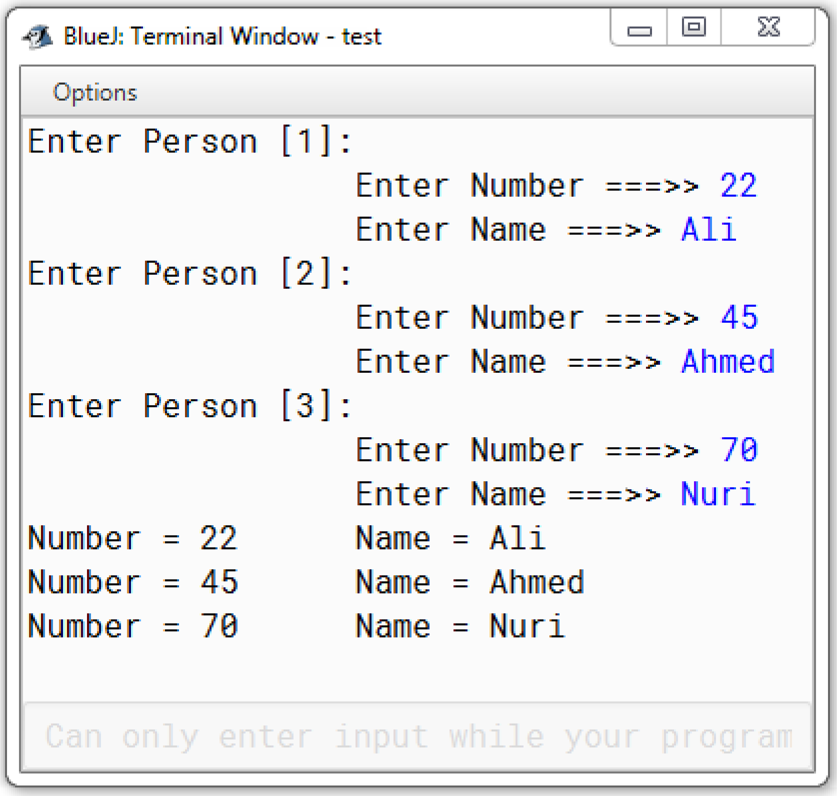
طباعة رقم الشخص المخزن بالمصفوفة في الموقع i

طباعة اسم الشخص المخزن بالمصفوفة في الموقع i

المصفوفة السابقة يمكن تمثيلها كالتالي:



وعند التنفيذ ستكون النتيجة كالتالي:



لاحظ أن البرمجة السابقة ستغنيك عن إنشاء مصفوفتين منفصلتين، الأولى نوعها صحيح وتمثل أرقام الأشخاص، والثانية مصفوفة من السلاسل النصية تمثل الأسماء).

```
final int MAX = 3;
int numbers = new int[MAX];
String names = new String[MAX];
```

المصفوفات ثنائية البعد Two Dimensional Array

في المصفوفات الثنائية البعد يتم تخزين البيانات على هيئة صفوف وأعمدة، ويمكن الوصول لأي عنصر من عناصر المصفوفة عن طريق رقم الصف ورقم العمود.

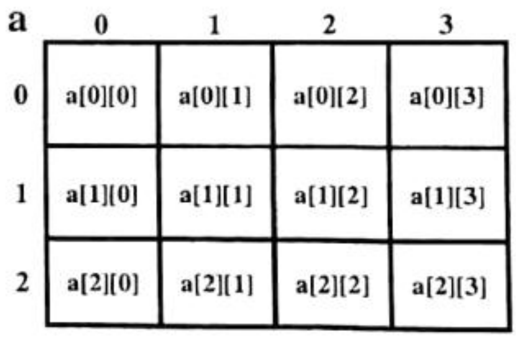
الشكل العام لتعريف مصفوفة ثنائية البعد هو:

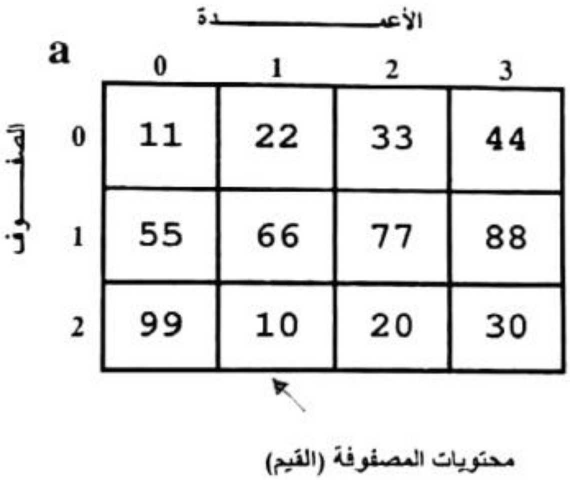
```
Type [][] matrixName = new Type[int-exp][int-exp];
```

فلتعريف مصفوفة اسمها a مكونة من ثلاثة صفوف وأربعة أعمدة من الأعداد الصحيحة نكتب:

```
int[][] a = new int [3] [4];
```

بصورة بسيطة يمكن تخيلها:





فللوصول للعنصر الذي في الصف رقم 1 والعمود رقم 2 وتخزينه في متغير أخر اسمه k نكتب:

```
k = a [1] [2];
```

يعني أن k الآن يحتوي على القيمة 77.

أما لو أردنا أن نخصص قيمة لعنصر من عناصر المصفوفة فنكتب:

```
a[2][3] = 40;
```

يعني خزن القيمة 40 في موقع المصفوفة ذو الصف رقم 2 والعمود رقم 3، فتصبح كالتالي:

	0	1	2	3
0	11	22	33	44
1	55	66	77	88
2	99	100	120	40

المصفوفة الثنائية البعد يمكن النظر إليها على أنها مصفوفة أحادية، وكل خلية بها هي مصفوفة أحادية أخرى.

فإذا كان لدينا التعريف التالي:

```
int [] [] a = new int [3] [4];
```

فيمكننا إيجاد عدد الصفوف بالمصفوفة كالتالي:

```
a.length
```

والتي سترجع طول البعد الأول للمصفوفة، أي 3، أما إذا أردنا إيجاد البعد الثاني للمصفوفة (أي عدد الأعمدة) فيمكننا كتابة التالي:

```
a[1].length
```

أي أوجد حجم الصف الأول (بمعنى أخر: عدد الأعمدة بالصف الأول وهي هنا 4).

في المصفوفات الثنائية عادة ما تكون كل الصفوف متساوية الطول كما سبق، ولكن في الجافا يمكن الإعلان عن مصفوفة ذات صفوف مختلفة الطول، فمثلا يمكن تكوين مصفوفة بشكل مثلثي كالتالي:

```
a[0][0]
a[1][0] a[1][1]
a[2][0] a[2][1] a[2][2]
a[3][0] a[3][1] a[3][2] a[3][3]
a[4][0] a[4][1] a[4][2] a[4][3] a[4][4]
```

فلتعريف مصفوفة كالسابق يمكن كتابة التعريف التالي:

```
int [] [] a = new int [5][ ];
```

أولا: قمنا بتحديد عدد الصفوف فقط كما سبق ولم نحدد عدد الأعمدة (طول كل صف).

ثانيا: سنقوم بتحديد طول كل صف لوحده كالتالي:

```
for (int i = 0; i < a.length ; i++)
a[i] = new int[i+1];
```

حيث أن الحلقة ستبدأ من 0 إلى عدد الصفوف (a.length) ثم في كل صف سنقوم بتحديد طوله (عدد العناصر التي ستخزن به) باستخدام:

```
a [i] = new int[i+1];
```

فمثلا في الصف الأول (عندما i=0) سيكون طول الصف 1 (i+1 أي 0+1) وهكذا لبقية الصفوف.

مثال:

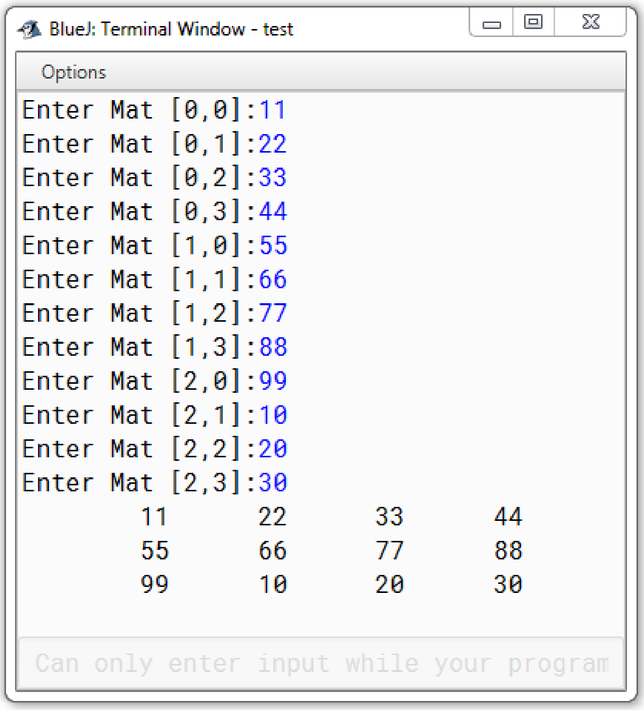
اكتب برنامج جافا مكون من فصيل اسمه *Matrix* يحتوي فقط على الدالة الرئيسية *main*() لإدخال قيم صحيحة في مصفوفة ثنائية البعد 4x3 وطباعتها على شكل مصفوفة ثنائية.

```
import java.util.Scanner;
 public class Matrix
 {
     public static void main(String[] agrs)
     {
         final int ROWS = 3;
         final int COLS = 4;
         int[] [] mat = new int[ROWS] [COLS];

         Scanner kb = new Scanner(System.in);
         System.out.print ('\u000C');
```

```
        // إدخال بيانات المصفوفة
        for(int i = 0 ; i< ROWS ; i++)
        {
            for (int j = 0 ; j < COLS; j++)
            {
                System.out.print("Enter Mat [" +i+ "," +j+ "]:");
                mat[i][j] = kb.nextInt();
            }
        }

        //طباعة عناصر المصفوفة
        for(int i = 0 ; i < mat.length ; i++)
        {
            for (int j = 0 ; j < mat[i].length; j++)
               System.out.print ("\t" + mat[i][j] );
            System.out.print('\n');
        }
    }
}
```

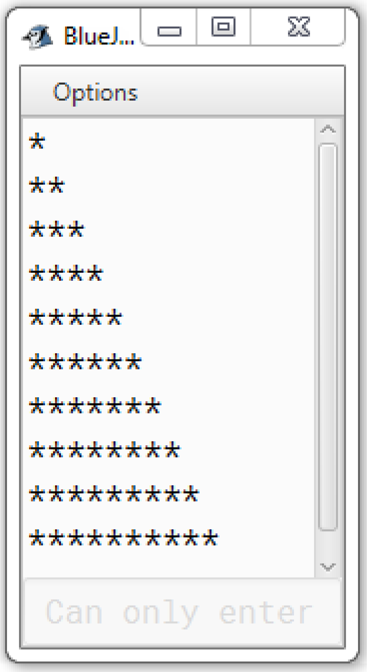


في طباعة عناصر المصفوفة يمكننا استخدام حلقة for السابقة مع حقل الطول length أو عدد الصفوف والأعمدة مباشرة كالتالي:

```
for (int i = 0 ; i< ROW ; i++)
{
    for (int j =0 ; j < COLS; j++)
        System.out.print ("\t"+mat[i][j]);
    System.out.print('\n');
}
```

مثال:

اكتب برنامج جافا مكون من فصيل اسمه: ***TriangularSahpe***، يحتوي فقط الدالة الرئيسية ***main( )*** لتكوين مصفوفة ثنائية البعد بحجم 10x10 لتخزين رمز"*" بها على شكل مثلثي ومن ثم طباعتها كالتالي:




```
import java.util.Scanner;
public class TriangularShape
{
   public static void main (String[] args)
   {
      final int ROWS = 10;
```

```
        final int COLS = 10;

        // عند تكوين المصفوفة؛ قمنا فقط بتحديد عدد الصفوف
        char[][] a = new char[ROWS][];

        Scanner kb = new Scanner (System.in);

        // هنا قمنا بتحديد طول كل صف (عدد الأعمدة بكل صف)
        for (int i = 0 ; i< a.length ; i++)
            a[i] = new char [i+1];

        System.out.print('\u000C');

        for (int i = 0 ; i< ROWS ; i++)
            for (int j =0; j < a[i].length; j++)
                a[i][j] = '*';

        for (int i = 0 ; i< ROWS; i++)
        {
            for (int j =0 ; j < a[i].length; j++)
                System.out.print(a[i][j]);
            System.out.print ("\n");
        }
    }
}
```

كن حذرا عند التعامل مع طول كل صف لوحده، أي عند التعامل مع الأعمدة بكل صف لوحده

مثال:

اكتب برنامج جافا مكون من فصيل اسمه *Matrix2* يحتوي فقط على الدالة الرئيسية **main()** لإدخال قيم صحيحة في مصفوفة ثنائية البعد 3x3 وتخزين مجموع كل صف في مصفوفة أحادية البعد ثم طباعة عناصر المصفوفة الثنائية وبجوارها عناصر المصفوفة الأحادية.

```
import java.util.Scanner;
public class Matrix2
{
    public static void main (String[ ] agrs )
    {
        final int ROWS = 3;
        final int COLS = 3;
        int [ ] [ ] mat = new int [ROWS][COLS];
        int [ ] arr = new int [ROWS];
        int sum;

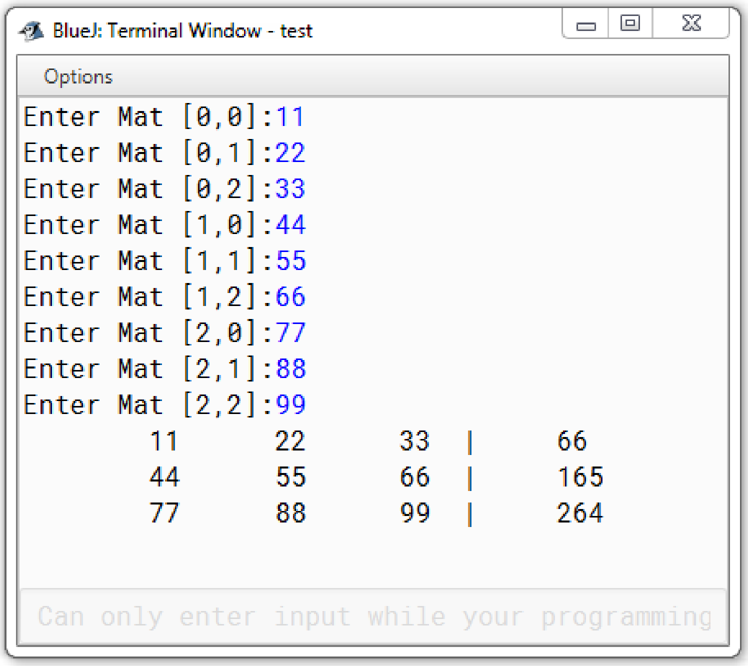
        Scanner kb = new Scanner(System.in);
        System.out.print ('\u000C');

        //إدخال البيانات للمصفوفة
        for (int i = 0 ; i < ROWS; i ++)
            for (int j = 0; j < COLS; j++)
            {
                System.out.print("Enter Mat [" +i+ "," +j+
"]:");
                mat [i][j] = kb.nextInt();
            }
```

```
        //معالجة/إيجاد مجموع كل صف وتخزينه في المصفوفة الأحادية
        for(int i = 0; i< ROWS; i++)
        {
            sum = 0;
            for (int j = 0 ; j < COLS; j++)
                sum = sum + mat[i][j];
            arr[i] = sum;
        }

        //طباعة عناصر المصفوفة الثنائية وبجوارها الأحادية
        for(int i = 0; i< mat.length; i++)
        {
            for(int j = 0 ; j < mat[i].length ; j++)
                System.out.print("\t" + mat[i][j]);
            System.out.print("   |      " + arr[i] + "\n");
        }
    }
}
```

النتيجة ستكون كالتالي:



Exceptions — الإستثناء

لاحظ أنه عند محاولة الوصول لعنصر من عناصر المصفوفة خارج حدودها (أكبر من أو يساوي طولها) فسوف يحدث استثناء (أي حدوث خطا عند التنفيذ ***runtime***)، فمثلا إذا كان لدينا:

```
int [] a = new int[5];
```

فإذا كتبنا حلقة for للوصول إلى عناصر المصفوفة كالتالي:

```
 for ( int i = 0 ; i <= 5 ; i++)
     System.out.println(a[i]);
```

لاحظ هنا أننا كتبنا (**i <= 5**) في حين أن أخر فهرس في المصفوفة هو 4، فإن الإستثناء الذي سوف يحدث اسمه: `Array IndexOutOfBondsException` (أي محاولة الوصول لفهرس خارج حدود المصفوفة).

أو عند محاولة القسمة على الصفر مثلا فإنه يحدث استثناء اسمه ArithmeticException سوف يحدث فمثلا:

```
int b = 99/0;
```

يمكننا معالجة الاستثناءات باستخدام جملة try...catch كالتالي:

```
try
{
     جمل من المحتمل أن تطلق استثناءات;
}
catch(Exception-name e)
{
     الإجراء المتخذ عند حدوث الإستثناء;
}
```

فمثلا إذا كان لدينا التعريف الآتي:

```
int [] a = {10, 20, 30, 40, 50};
try
{
     for(int i = 0; i <= 5; i++)
          System.out.println(a[i]);
}
catch( ArrayIndexOutOfBoundsException e)
{
```

عند محاولة الوصول إلى فهرس عنصر خارج حدود المصفوفة فسيتم إطلاق استثناء

```
    System.out.println("Error: index out of bounds");
}
```

إذا انطلق الإستثناء فإن جملة ***catch*** سوف نمسكه وتظهر هذه الرسالة.

يمكننا كتابة أكثر من جملة catch لإلتقاط الإستثناءات كالتالي:

```
int [] a = {10, 20, 30, 40, 50};
try
{
    //محاولة القسمة على صفر
    int b = 99/0;

    محاولة الوصول لفهرس خارج حدود المصفوفة
    for(int i = 0; I <= 5; i++)
        System.out.println(a[i]);
}

// معالجة استثناء محاولة الوصول لفهرس خارج حدود المصفوفة
catch ( ArrayIndexOutOfBoundsException e)
{
    System.out.println("Error: index out of boundary");
}

// معالجة استثناء محاولة القسمة على صفر
catch (ArithmeticException e)
{
    System.out.println("Error: Division by Zero");
}

//  معالجة أي استثناء أخر لم يتم التقاطه ضمن جمل الإلتقاط السابقة
catch (Exception e)
{
    System.out.println("Error: Unknown Error ");
}
```

e: هو كائن من نوع الإستثناء يمكن استخدامه لطباعة رسالة تبيين نوع الإستثناء الذي حدث باستدعاء الطريقة `getMessage()` كالتالي:

```
try
{
    int b = 50/0;
}
```

كائن من نوع Exception

هذا المنهج سوف يظهر رسالة تبين فحوى الإستثناء الذي حدث، وهو هنا: القسمة على صفر

```
catch(Exception e)
{
    System.out.println("Error: " + e.getMessage() ) ;
}
```

كما يمكن استخدام المنهج **e.toString()** ليظهر اسم فصيل الإستثناء مع فحوى رسالة تبين ما هو الخطأ الذي حدث.

كما يمكننا استخدام جملة **finally** والتي ستؤدي لتنفيذ جمل برمجية ضمنها عند حدوث أو عدم حدوث أي استثناء ضمن جمل الـ **try**. فعند استخدام **catch … try** وحدوث استثناء ضمن إحدى جمل الـ**try** فإن التنفيذ سوف يتوقف عند تلك الجملة التي حدث بها الخطأ ويتجه لـ **catch** مباشرة لمعالجة الإستثناء، أما عند استخدامنا لجملة **finally** فإن الجمل البرمجية سوف تنفذ ضمنها بغض النظر عن حدوث ومعالجة الإستثناء، أو عدم حدوثه. والشكل العام لها:

```
try
{
    جمل من المحتمل أن تلقي استثناءات;
}
finally
{
    جمل سيتم تنفيذها سواء حدث استثناء أم لم يحدث;
}
```

عادة ما يتم استخدام **finally** للقيام بإجراءات يجب أن تتم قبل إنهاء البرنامج سواء حدث الإستثناء (الخطأ الذي سيوقف البرنامج) أم لم يحدث؛ مثل إغلاق الملفات أو إرجاع موارد ذاكرة تم حجزها مسبقا قبل إنهاء البرنامج، سواء حدث الإستثناء أم لم يحدث.

يمكن استخدام جملة **finally** بعد **catch** في جملة **try** واحدة.

لا يفوتنا أن نذكر أن الإستثناءات (وكما موضح بالشكل التالي) يتم تمثيلها ضمن الجافا بفصائل مختلفة ترث الفصيل الأعلى **Throwable** وأن هناك فصيلتين ترث هذه الفصيلة، هما: **Exception** و **Error** وأن الإستثناءات "**Exception**" تنقسم لنوعين هما:

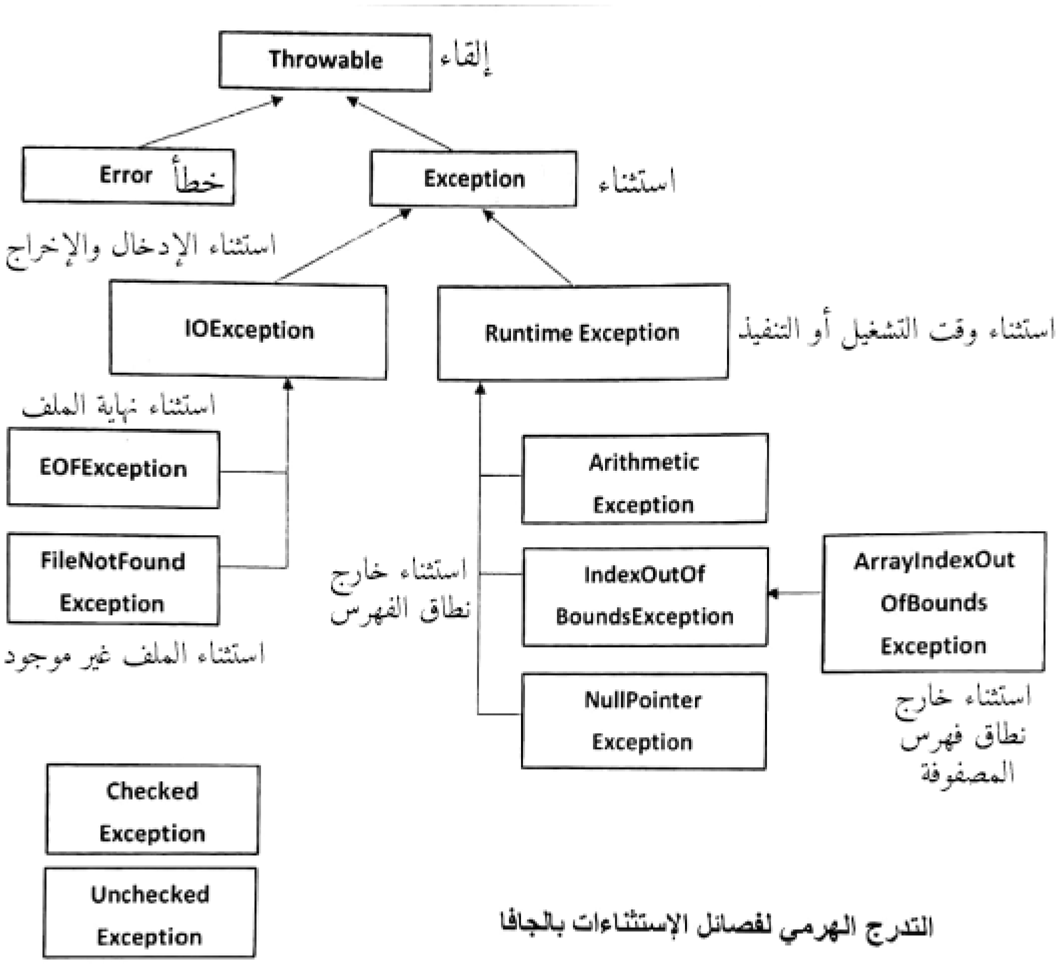
1- Checked Exception: وهي التي لا يمكن إهمالها، مثل: كل الإستثناءات المتعلقة بالإدخال والإخراج I/O.
2- Unchecked Exception: وهي الإستثناءات التي ليس من الضروري أن تكون معالجة كاستثناءات وقت التنفيذ (مثلا: **ArithmeticException**).

كما يمكننا تكوين فصائل خاصة بنا للإستثناءت تورث كل خصائص ومناهج فصيل الإستثناء "Exception".

وبالتالي يمكننا إطلاق هذه الإستثناءات (باستخدام جملة **throw**) ومعالجتها بجملة **catch** أو تمريرها (باستخدام جملة **throws**) لفصائل أخرى لتقوم بمعالجتها.

يمكننا تلخيص ما سبق بالتالي:

إن الإستثناء هو خطأ يحدث أثناء البرنامج فيؤدي لتوقف التسلسل الطبيعي للتنفيذ.



التدرج الهرمي لفصائل الإستثناءات بالجافا

مثال:

اكتب برنامج جافا مكون من فصيلين، الأول اسمه MyException لتعريف استثناءك الخاص حيث يرث فصيل الإستثناء Exception ويحتوي على باني واحد يقوم باستقبال رسالة مطلوب إظهارها عند حدوث الخطأ وإرسالها للفصيل الأعلى (Exception) بال super.

أما الفصيل الأخر UseMyException ويحتوي فقط على الدالة الرئيسية ()main فيقرأ قيمة صحيحة ويختبرها، إذا كانت صفر يقوم البرنامج بإطلاق استثناءك الخاص ومعالجة الإستثناء عن طريق إلتقاطه بـ catch وإظهار رسالة الإستثناء، مع توضيح استخدام جملة finally.

الفصيل الأول MyException:

```
import java.util.Scanner;
public class UseMyException
{
    public static void main (String[] args)
```

```
    {
        Scanner kb = new Scanner(System.in);
        try
        {
            System.out.print('\f');

            System.out.print("Enter A : ");
            int a = kb.nextInt();

            if(a == 0)
                throw new MyException("No Division by zero ");

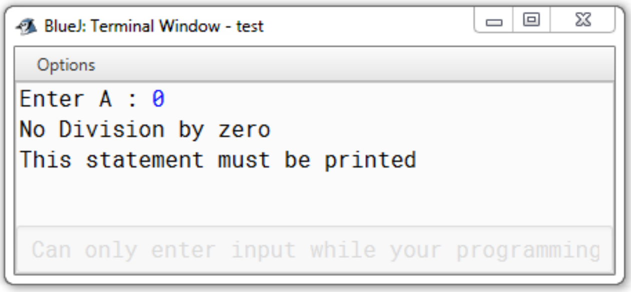
            int b = 20/a;
            System.out.println("Result = " + b);
        }
        catch (MyException e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.print("This statement must be printed");
        }
    }
}
```

هنا يتم اختيار القيمة المدخلة، إذا ساوت صفر يتم إلقاء استثناء بجملة **throw** مع إرسال الرسالة المطلوب إظهارها

إذا تم التقاط الإستثناء الذي ألقي بواسطة جملة **throw** السابقة فسوف يتم طباعة الرسالة المرسلة عن طريق المنهج **getMessage()**

هذه الجملة سيتم طباعتها في جميع الأحوال.

لاحظ أنه قد استخدمنا جملة الإلقاء (**throw**) عند إدخال قيمة صفر، لأنه لايمكن القسمة على الصفر، لذلك قمنا بحذف كائن جديد من نوع الإستثناء الذي قمنا بتكوينه باستخدام **new** مع إرسال الرسالة التي سوف يتم إظهارها، هذه الرسالة ("No Division by zero") سيتم استقبالها بواسطة المعلمة "parameter" **m** ضمن باني الفصيل السابق (**MyException**) والذي سيُرسل بدوره للفصيل الأعلى (**Exception**) بـ **super**. ولكون الإستثناء حدث ضمن جملة **catch** … **try** فسنقوم بالتقاط الإستثناء باستخدام جملة catch ومن ثم طباعة الرسالة باستخدام المنهج **getMessage()**. أما الجملة ضمن الـ **finally** فسيتم طباعتها في جميع الأحوال.



تـدريـبات

1- اكتب برنامج جافا مكون من فصيل اسمه ***MaxInArray*** يحتوي فقط على الدالة الرئيسية ***main()*** لإدخال سبعة عناصر صحيحة في مصفوفة أحادية وطباعة أكبر عنصر في المصفوفة.

2- اكتب برنامج جافا مكون من فصيل اسمه MyShape يحتوي فقط على الدالة الرئيسية ***main()*** يستخدم مصفوفة ثنائية البعد 5x5 عند طباعتها ينتج الشكل التالي:

```
* - - - *
- * - * -
- - * - -
- * - * -
* - - - *
```

3- اكتب برنامج جافا مكون من فصيل اسمه: SeatsBooking، يحتوي فقط على الباني والذي يقوم بفتح رحلة جديدة لحجز الركوب بطائرة مكونة من 10 صفوف وأربعة أعمدة من المقاعد، ويحتوي أيضا على منهج اسمه: **printSeats()**، يقوم بطباعة مقاعد الطائرة المحجوزة والغير محجوزة، أيضا يحتوي على منهج اسمه: **bookseat()**، يقوم بحجز مقعد للزبون وطباعة الـok في حالة وجود مقعد فارغ وحسب رغبة الزبون أو رسالة تدل على أن المقعد المختار مشغول ولايمكن حجزه.

4- اكتب برنامج جافا مكون من فصيل اسمه AvgMaxMinArray يحتوي فقط على الدالة الرئيسية **main()** لإدخال ستة عناصر صحيحة في مصفوفة أحادية وطباعة المتوسط الحسابي لأكبر وأصغر عنصرين في المصفوفة.

5- اكتب برنامج جافا مكون من فصيل اسمه Finding يحتوي فقط على الدالة الرئيسية **main()** لإدخال عشرة عناصر صحيحة في مصفوفة أحادية البعد وطباعة موقع أول عنصر سالب وكذلك موقع أول عدد زوجي.

6- اكتب برنامج جافا مكون من فصيل اسمه Average يحتوي فقط على الدالة الرئيسية **main()** لإدخال ثمانية عناصر في مصفوفة أحادية البعد وطباعة معدل الأرقام الفردية وحاصل جمع الأعداد الزوجية وعدد الأرقام السالبة.

7- اكتب برنامج جافا مكون من فصيل اسمه SumColumns يحتوي فقط على الدالة الرئيسية **main()** لإدخال قيم صحيحة في مصفوفة ثنائية البعد 3x3 وطباعتها على شكل مصفوفة ثم طباعة مجموع كل عمود لوحده أسفل المصفوفة.

8- اكتب برنامج جافا مكون من فصيل اسمه PosNegMatrix يحتوي فقط على الدالة الرئيسية **main()** لإدخال بيانات صحيحة في مصفوفة حجمها 4x4 وإيجاد مجموع القيم الموجبة ومجموع القيم السالبة، ومن ثم الفرق بين مجموع القيم والموجبة والقيمة المطلقة لمجموع القيم السالبة.

9- اكتب برنامج جافا مكون من فصيل اسمه SumPos2AndMin4 يحتوي فقط على الدالة الرئيسية **main()** لإدخال بيانات عددية في مصفوفة ثنائية البعد بحجم (4x5) وأوجد مجموع العناصر الموجبة ضمن الصف الثاني، وأصغر قيمة في الصف الرابع.

مقدمة عن هياكل البيانات

قبل أن نتعرف على هياكل البيانات، علينا أولا التعرف على كل من البيانات والمعلومات ومالفرق بينهما.

البيانات "data": هي مجموعة من الحقائق عن شيء ما لم يتم معالجتها، مثل: الأرقام أو الحروف أو الصور أو خليط بينها، فإذا تمت معالجته بطريقة معينة لتصبح ذات فائدة أصبحت **معلومات** "information"، بمعني أن **البيانات:** هي المادة الخام للمعلومات، وأن الفرق الأساسي بينهما في المعالجة. المعالجة قد تأخذ أنواع عدة، مثلا: الإضافة، الحذف، الدمج، الترتيب، التعديل، التخزين والإسترجاع وغيرها.

أكثر ما يعنينا هنا هو كيفية تنظيم البيانات في هياكل محددة لتسهيل المعالجة (وبالأخص عمليتي التخزين و الإسترجاع)، فكلما كانت هاتين العمليتين أكثر تعقيدا، تطلبتا زمنا أطول للمعالجة، وأن كل من عمليتي التخزين والإسترجاع تعتمد بشكل مباشر على نوع بنية أو هيكل البيانات. **فما هي هياكل البيانات؟**

في الحقيقة أن الهيكلية هي خاصية أساسية من خصائص البيانات. فمثلا إذا نظرنا من حولنا نجد أن هناك العديد من الأمثلة الواقعية عن هياكل البيانات، فمثلا:

- ❖ **شجرة العائلة،** نجد أن الجد يكون متمركزاً في الجذر، يليه الأبناء، ثم الأحفاد في نهاية الشجرة كأوراق.
- ❖ **طابور شراء الحاجيات،** حيث نجد أن الشخص الواقف أولا متمركزاً في بداية الطابور، والشخص القادم أخيراً في ذيل الطابور.
- ❖ **جدول مواعيد إقلاع ووصول الطائرات في رحلات الطيران.**
- ❖ **قوائم أسماء الطلبة المرتبة على حسب الأسماء أو أرقام القيد.**
- ❖ **المخططات والخرائط،** فمثلا مخطط للسفر من مدينة لأخرى مرورا بعدة مدن سالكين أقصر مسار.
- ❖ **دليل الهاتف،** الحاوي على أسماء الأشخاص/الجهات وأرقام الهواتف والمرتبة أبجديا.

وهناك العديد والعديد من الأمثلة على البيانات وهيكلتها، فهياكل البيانات هي باختصار طرق تنظيم البيانات بغية توفير المعالجة المطلوبة. أو بمعنى آخر هي عبارة عن أليات وخوارزميات تعمل على تنظيم ترابط البيانات مع بعضها البعض في الذاكرة بحيث تأخذ هيكلا معيناً، بغية الإسراع في الوصول للبيانات ومعالجتها من أجل أداء غرض محدد.

فهيكلة البيانات لها العديد من الفوائد، مثلا:

- ❖ سرعة تخزين واسترجاع البيانات من الذاكرة.
- ❖ استغلال الذاكرة بشكل جيد وفعال.
- ❖ المساعدة في إنشاء برامج قوية من حيث البناء والمنطق.
- ❖ مساعدة المبرمجين في كتابة البرامج المختلفة بطرق مبتكرة ومرنة.

لماذا هناك العديد من هياكل البيانات؟

هيكل البيانات النموجي هو الذي يحقق السرعة والبساطة ويستخدم الذاكرة بطريقة جيدة وفعالة. فعند استخدام إحدى الهياكل من هياكل البيانات عليك أن تفاضل في اختيار الهيكل المناسب بعناية بناءً على ما يقدمه هذا الهيكل، فمثلا:

- ❖ الوقت في مقابل المساحة التخزينية.
- ❖ العمومية في مقابل البساطة.
- ❖ أداء عملية معينة في مقابل أداء عملية أخرى.

وهناك العديد من الأمور التي ينبغي عليك الموازنة فيما بينها عند استخدامك للبيانات. فدراسة هياكل البيانات تُعنى بدراسة المفاضلة بين ما ذكر، وهذا بطبيعة الحال نتج عنه تنوع في استخدام هياكل البيانات.

هياكل البيانات تعتمد بشكل أساسي على (وترتبط ارتباطا وثيقا بـ) الخوارزميات. **فما هي الخوارزمية؟**

الخوارزمية: هي وصف لمجموعة من الخطوات المنطقية ذات مستوى عالي والتي يتم تنفيذها حسب ترتيب معين لحل مسألة ما، وتكون مستقلة عن أية لغة برمجة. فالخوارزمية ينبغي أن تكون مختصرة الخطوات وبسيطة ودقيقة، كما ينبغي أن تكون قابلة للتطبيق باستخدام لغات البرمجة، والأهم من ذلك أن تكون سريعة التنفيذ ورخيصة الثمن عند تنفيذها على الحاسوب.

أنواع هياكل البيانات

1- هياكل البيانات الثابتة.

2- هياكل البيانات المتغيرة.

1 هياكل البيانات الثابتة fixed-size data structures

و هي التي عدد عناصرها ثابتة الحجم ولا تتغير، لأنها لاتقبل عمليتي الإضافة والحذف، مثل: المتجهات، المصفوفة، والسجلات.

2 هياكل البيانات المتغيرة dynamic-size data structures

هي التي عدد عناصرها في حالة زيادة ونقصان مستمر طيلة حياة الهيكل نتيجة عمليتي الإضافة والحذف، و هما نوعان:

أ- **هياكل بيانات متغيرة خطية:**

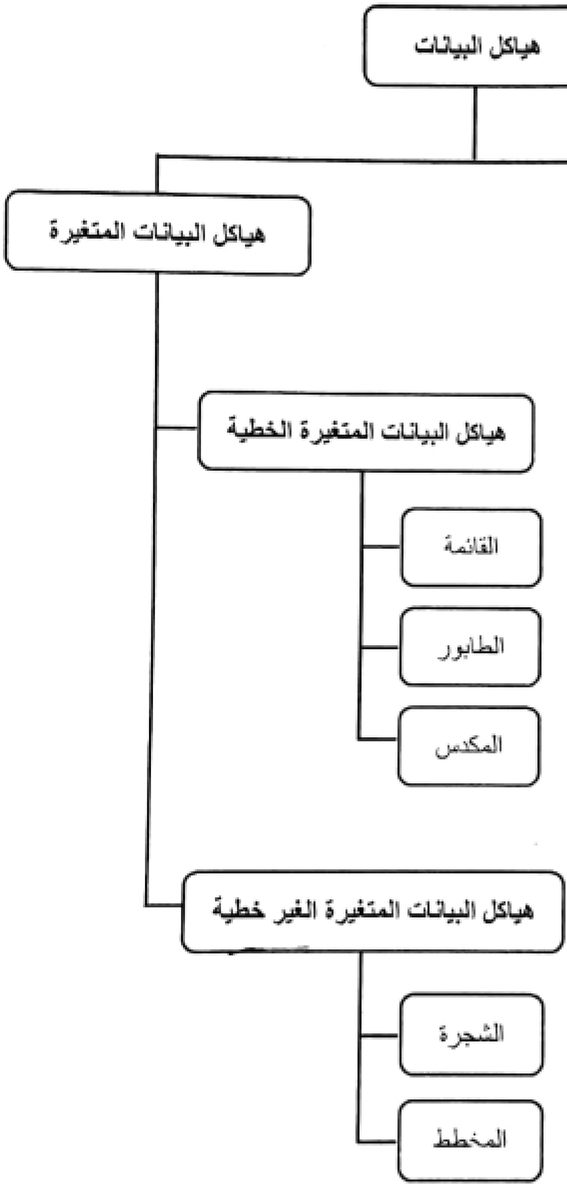
هي التي تنتظم عناصرها في خط متتالي بحيث أن كل عنصر في الهيكل يليه عنصر واحد فقط ما عدا الأخير، وأيضا كل عنصر يسبقه عنصر واحد فقط عدا الأول، ومن أمثلة هذه الهياكل:

القائمة (List)، الطابور (Queue)، المكدس (Stack)، الملف (File) وغيرها.

ب- **هياكل بيانات متغيرة غير خطية:**

وهي التي تتشعب عناصرها في أكثر من اتجاه، بمعنى أنه يسمح للعنصر بأن يليه أكثر من عنصر مثل الشجرة (tree)، وقد يسمح للعنصر أن يسبقه ويليه أكثر من عنصر أخر مثل المخطط (graph).

المخطط التالي يوضح أنواع هياكل البيانات المختلفة.



في الحقيقة أن هناك ميزتان تصاحب عملية تنظيم البيانات في هيا كل مختلفة هما:

الترتيب والقيمة: أن كل عنصر من عناصر البيانات داخل الهيكل له قيمة معينة قد تكون صحيحة أو حقيقية أو منطقية أو كائنات من فصيل معين، كما أن كل العناصر داخل الهيكل تكون منظمة في ترتيب معين بحيث يمكن الوصول لعنصر معين بترتيبه نسبة لباقي العناصر، فإذا اختل هذا الترتيب فإن الهيكل قد يأخذ شكلاً آخر، فمثلا في هيكل الشجرة يكون لكل عنصر أب واحد فقط عدا الجذر أي يسبق كل عنصر عنصر واحد فقط، فعندما نضيف فرع (edge) جديد لأحد الأبناء بحيث يصبح متفرع من عقدتين سيتغير الهيكل من شجرة لمخطط. وكذلك الحال بالنسبة للقائمة، فإذا قيدنا عمليتي الإضافة و الحذف لتكون من جهة واحدة فقط سيتحول الهيكل إلى مكدس.

اختلاف التمثيل الفيزيائي عن التمثيل المنطقي داخل ذاكرة الحاسوب:

من الناحية المنطقية، يفترض عند تخزين عناصر البيانات المدخلة في ذاكرة الحاسوب أن تكون متتالية كما تم إدخالها، ولكن من الناحية الواقعية أو الفيزيائية، يتم تخزينها على حسب الأماكن الفارغة في الذاكرة، بمعنى أن عناصر الهيكل الواحد قد تكون موزعة في أماكن متناثرة وغير متجاورة، ولكن تربطها مؤشرات تدل على العناصر اللاحقة بها.

المصفوفات الأحادية من ناحية أخرى عادة ما يتم تخزينها في أماكن متتالية في الذاكرة. فمثلا إذا كان لدينا مصفوفة من m من العناصر، وأن عناصرها هو من النوع الصحيح القصير بحيث أن كل عنصر يأخذ 2 بايت، وأن عنوان أول عنصر ضمن المصفوفة تم حجزه في الذاكرة يبدأ من 100، فيمكن تمثيل المصفوفة التي تم حجزها في الذاكرة على النحو التالي:

العنوان في الذاكرة	بايت 1	بايت 2
100		
102		
104		
106		
.		
.		
100+(2*(m-1))		

ولكن المبرمج عادة ما يستخدم تمثيلا منطقيا أخر للمصفوفة، وبعيدا عن حساب العنونة في الذاكرة. بحيث يستخدم فهرسا يبدأ عادة من 0، 1، 2، ....، **m-1**، وأن لغة البرمجة ستقوم بتحويل الفهرس المنطقي إلى ما يقابله من عناوين فيزيائية ضمن الذاكرة، فيمكن تصور المصفوفة الأحادية من وجهة نظر المبرمج كالتالي:

0	1	2	3	.	.	m-1

أما المصفوفات الثنائية ولتكن **$m * n$** فيمكن اعتبارها كعدد **n** من المصفوفات الأحادية كل منها لها **m** من العناصر إذا تم تخزينها في الذاكرة بناء على الصفوف (أي صفا بعد الأخر: ***row-major***)، أو يمكن اعتبارها كعدد **m** من المصفوفات الأحادية كل منها لها عدد **n** من العناصر إذا تم تخزينها في الذاكرة على حسب الأعمدة (أي عمود تلوى الأخر ***column-major***).

فمثلا إذا كان لدينا المصفوفة التالية A من رتبة 4x3، فيمكن تصور تخزينها في الذاكرة كالتالي:

	0	1	2	3
0	11	22	33	44
1	55	66	77	88
2	99	100	101	102

تمثيل المصفوفة داخل الذاكرة صفا تلو الأخر بالمتجه B:

صف 1				صف2				صف3			
11	22	33	44	55	66	77	88	99	100	101	102
0	1	2	3	4	5	6	7	8	9	10	11

أما تمثيل المصفوفة داخل الذاكرة عمودا تلو الأخر بالمتجه C فيكون:

عمود 1			عمود 2			عمود 3			عمود 4		
11	55	99	22	66	100	33	77	101	44	88	102
0	1	2	3	4	5	6	7	8	9	10	11

ولكن ماهي العلاقة الرياضية التي تربط عناصر المصفوفة الثنائية A بكل من عناصر المتجهين B و C.

بشكل عام إذا كانت المصفوفة A تحتوي على عدد من الصفوف m وعدد من الأعمدة n وأن هناك فهرسان (دليلان) هما i و j تحددان موقع كل عنصر داخل المصفوفة A، حيث ان i يمثل فهرس الصف، وأن j يمثل فهرس العمود، فإن العلاقة التالية توجد موقع (*L*) كل عنصر من المصفوفة ***A*** داخل المتجه ***B***:

```
L = n*i+j
```

وأن العلاقة التي توجد موقع (*L*) كل عنصر من المصفوفة A داخل المتجه C:

```
L = m*j+i
```

مثال:

اكتب برنامج بلغة الجافا يقوم بإدخال أعداد صحيحة من لوحة المفاتيح في مصفوفة ثنائية البعد برتبة(4x3)، ثم طباعتها بشكل منطقي (أي في ثلاثة صفوف وأربعة أعمدة)، ثم تكوين وطباعة متجهين أخرين يمثلان التخزين الفيزيائي للمصفوفة داخل الذاكرة؛ أي بمعنى المتجه الأول يحتوي عناصر المصفوفة صفا بصف والثاني يحتوي على عناصر المصفوفة عمودا بعمود.

```
import java.util.Scanner;
public class LogicalAndPhysicalArray
{
    public static void main (String[] args)
    {
        final int ROWS = 3;
        final int COLS = 4;

        int [][] mat = new int [ROWS][COLS];

        Scanner kb = new Scanner (System.in);
        System.out.print('\u000C');

        for(int i = 0; i< ROWS ; i++)
            for(int j = 0; j < COLS; j++)
            {
                System.out.print("Enter Mat [" +i+ "," +j+ "] :
");
                mat[i][j] = kb.nextInt();
            }
        System.out.print("\n\n  Logical  representation  for
Matrix \n");

        for(int i = 0; i < mat.length; i++)
        {
            for(int j = 0; j < mat[i].length; j++)
                System.out.print ("\t" + mat[i][j] );
            System.out.print ('\n');
        }

        int k1, k2;

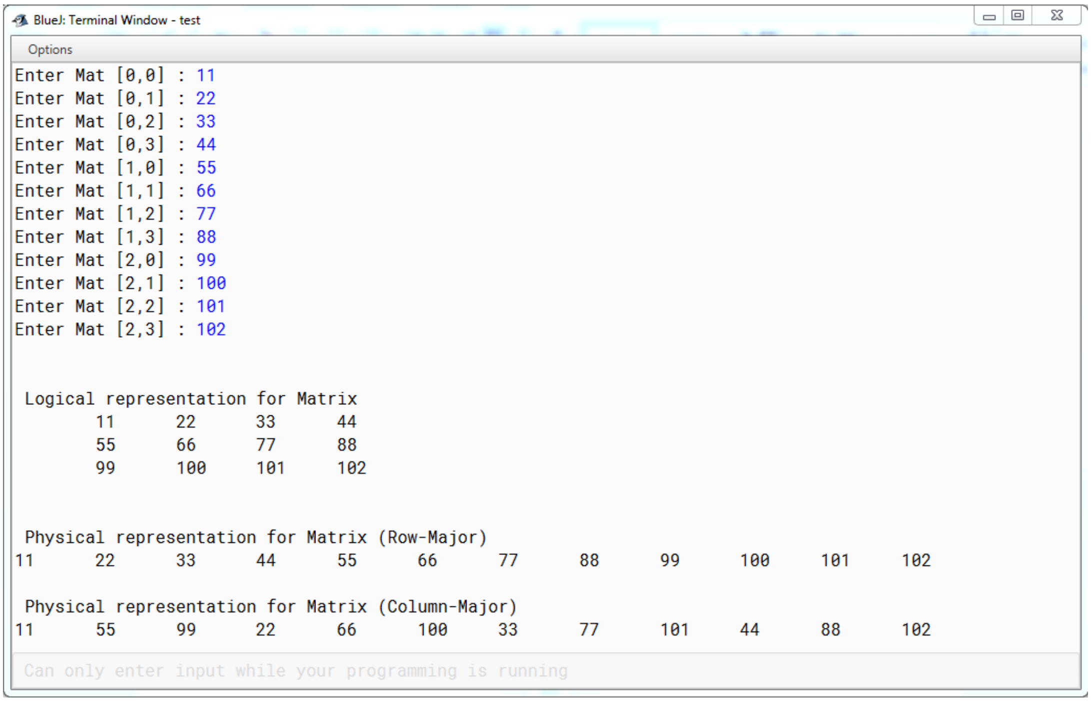
        int [] rowMajor = new int [ROWS * COLS];
```

```
        int [] columnMajor = new int [ROWS * COLS];

        for(int i = 0; i < ROWS; i++)
            for(int j = 0; j < COLS; j++)
            {
                k1 = COLS * i + j;
                rowMajor[k1] = mat[i][j];

                k2 = ROWS * j + i;
                columnMajor[k2] = mat[i][j];
            }
        System.out.print("\n\n  Physical  representation  for Matrix (Row-Major) \n");
        for(int i = 0; i < (ROWS * COLS); i++)
        {
            System.out.print(rowMajor[i] + "\t");
        }
        System.out.print("\n\n  Physical  representation  for Matrix (Column-Major) \n");
        for(int i = 0; i < (ROWS * COLS); i++)
        {
            System.out.print(columnMajor[i] + "\t");
        }
    }
}
```

النتيجة ستكون كالتالي:


BlueJ: Terminal Window - test
Options

```
Enter Mat [0,0] : 11
Enter Mat [0,1] : 22
Enter Mat [0,2] : 33
Enter Mat [0,3] : 44
Enter Mat [1,0] : 55
Enter Mat [1,1] : 66
Enter Mat [1,2] : 77
Enter Mat [1,3] : 88
Enter Mat [2,0] : 99
Enter Mat [2,1] : 100
Enter Mat [2,2] : 101
Enter Mat [2,3] : 102


 Logical representation for Matrix
        11      22      33      44
        55      66      77      88
        99      100     101     102


 Physical representation for Matrix (Row-Major)
11      22      33      44      55      66      77      88      99      100     101     102

 Physical representation for Matrix (Column-Major)
11      55      99      22      66      100     33      77      101     44      88      102
```

Can only enter input while your programming is running

القـائـمـة List

القائمة: هي هيكل من هياكل البيانات الخطية، فهي تمثل مجموعة من العناصر المنظمة في تسلسل. قد تكون هذه العناصر أعداد صحيحة أو حقيقية أوسلاسل نصية أو كائنات لأصناف معينة. عادة ما يتم تمثيل القائمة باستخدام إما المصفوفات (arrays) أو القوائم المترابطة (Linked list).

سنقوم بتمثيل القائمة باستخدام الطريقتين لإجراء عمليات المعالجة المختلفة منها على سبيل المثال:

- ❖ إضافة عنصر للقائمة.
- ❖ حذف عنصر من القائمة.
- ❖ تحديث أو تعديل عنصر في القائمة.
- ❖ البحث عن عنصر في القائمة واسترجاعه.
- ❖ زيارة عناصر القائمة بشكل تسلسلي (لأجل الطباعة مثلا).
- ❖ ترتيب عناصر القائمة.
- ❖ تقسيم قائمة لقائمتين.
- ❖ دمج قائمتين في قائمة واحدة.

أولا تمثيل القائمة باستخدام المصفوفات

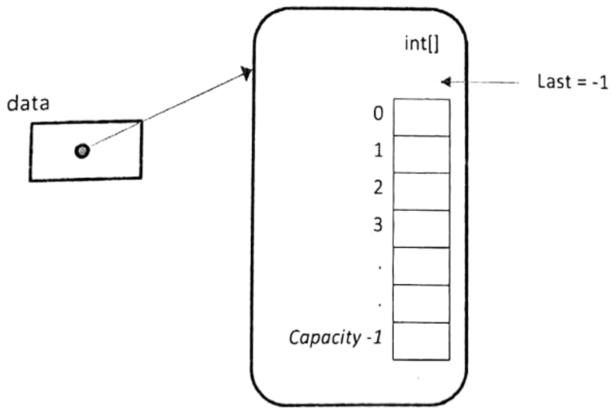
تستخدم المصفوفات لتخزين البيانات لما لها من سرعة الوصول لمواقع عناصرها، حيث يمكن الوصول لأي عنصر بشكل عشوائي ومباشر باستخدام فهرسه، غير أن عملية إضافة أو حذف عنصر في وسط المصفوفة قد تكون مكلفة حاسوبيا. إضافة إلى أنه قد يتم حجز مساحة من الذاكرة مسبقا للمصفوفة والتي قد لا يتم استغلالها بشكل فعال مما يؤدي لهدر موارد الحاسوب (كأن يتم من حجز 1000 موقع للمصفوفة لتخزين 10 عناصر فقط). بناء عليه، عليك المفاضلة بين هياكل البيانات قبل استخدامها. فإذا تطلبت عناصر البيانات لديك سرعة الوصول العشوائي (Ravictor Access) لعناصرها فعليك باستخدام المصفوفات، أما إذا كانت الإضافة والحذف تشكل الأهمية الأكبر لديك؛ فعليك باستخدام هيكل أخر، مثل: القائمة المترابطة، كما سيتم شرحه لاحقا.

قبل القيام بإجراء العمليات المختلفة على القائمة باستخدام المصفوفات سنقوم أولا بتكوين فصيل اسمه (*ListClass*) يحتوي على تعريف لمصفوفة من الأعداد الصحيحة اسمها (*data*) ذات حجم أو سعة (*capacity*) تحدد ضمن بانيها، بالإضافة لحقل اسمه (*last*) يدل على موقع آخر، عنصر بالقائمة كما يلي:

```
public class ListClass
{
    private int last;
    private int capacity;
    private int[] data;

    public ListClass(int c)
    {
        capacity = c;
        data = new int [capacity];
        last = -1;
    }
}
```

لاحظ أن حجم المصفوفة capacity يتم استقباله كمعلمة ضمن الباني، كذلك عند بناء المصفوفة لأول مرة، فإن الدليل last سيكون عند 1- وذلك لعدم وجود عناصر بالمصفوفة، هذا الدليل سيزداد بإضافة عناصر للمصفوفة وينقص مع عمليات حذف العناصر، فهو يدل على عدد البيانات الفعلية الخزنة بالمصفوفة على عكس المتغير `capacity` والذي يدل على الحجم الإجمالي للعناصر التي يمكن أن تسعها المصفوفة. هذا التعريف سيؤدي لتكوين مصفوفة فارغة من العناصر (إفتراضيا الجافا ستقوم بتعبئة المصفوفة بأصفار ولكن لتبسيط العملية سنعتبرها فارغة) كالتالي:



الآن يمكننا استخدام هذا الفصيل ضمن فصيل أخر وليكن (UseList) وتكوين كائن من القائمة كالتالي:

```
ListClass myList = new ListClass(5);
```

هنا حددنا سعة المصفوفة بـ 5 عناصر كحد أقصى. الرقم 5 سيمرر لباني الفصيل ListClass ليتم تخزينه ضمن الحقل capacity ثم تحديد طول المصفوفة بـ5.

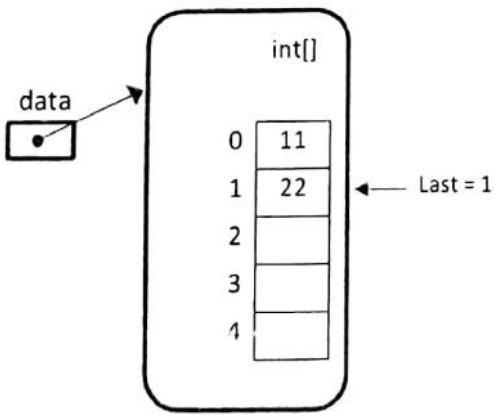
منهج الإضافة لنهاية القائمة add

الآن سنقوم بإضافة منهج جديد اسمه **add()** والذي سيقوم بإضافة عنصر جديد يتم استقباله كمعلمة لنهاية القائمة (المصفوفة). في حال ما إن كانت المصفوفة ممتلئة فسيتم إلقاء استثناء. مع كل عملية إضافة جديدة سيتم زيادة 1 لقيمة الحقل last.

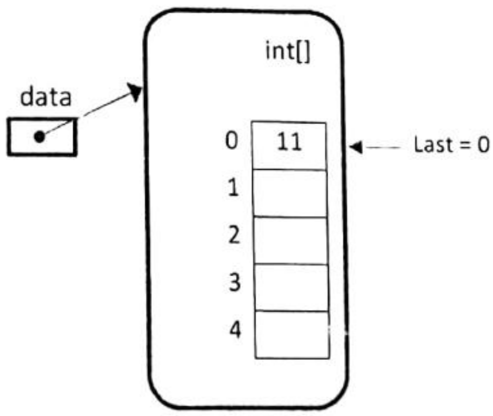
```
public void add(int d) throws Exception
{
    if (isFull())
        throw new Exception("(1) : The array is full !");
    else
    {
        last ++;
        data[last] = d;
    }
}
```

يمكننا استدعاء هذا المنهج الان ضمن الفصيل UseList عدة مرات وكالتالي:

```
myList.add(11);
myList.add(22);
```



القائمة بعد عملية الإضافة الثانية
myList.add(22)



القائمة بعد أول عملية إضافة
myList.add(11)

منهج هل القائمة ممتلئة isFull

لاحظنا من المنهج السابق أننا استخدمنا المنهج **isFull()** ليختبر ما إذا كانت القائمة ممتلئة أو لا. شرط الإمتلاء يتحقق عندما تكون قيمة **last** مساوية لـ 4 أي حجم المصفوفة الكلي ناقصا واحد (**capacity -1**) فالمنهج كالتالي:

```
public Boolean isFull()
{
     return (last == (capacity-1));
}
```

منهج هل القائمة فارغة isEmpty

تكون القائمة فارغة عندما يكون مؤشر الـ **last** أقل من الصفر، فالمنهج يكون كالتالي:

```
public Boolean isEmpty()
{
     return (last < 0);
}
```

منهج حشر أو إضافة عنصر في موقع معين بالقائمة insert

يمكن إضافة عنصر ما في موقع معين في بداية أو وسط القائمة؛ ولكن هذا سيؤدي بطبيعة الحال لإزاحة جميع العناصر الواقعة بعد هذا العنصر، والذي يعتبر مكلفا من حيث المعالجة. المنهج سيستقبل معلمتان: أحدهما العنصر المراد إضافته (**d**)، والثاني: الموقع المراد إضافته فيه (**index**).

لاحظ أنه عند محاولة الإضافة لقائمة فارغة أو الإضافة لآخر القائمة؛ سيتم استدعاء المنهج السابق **add()** حيث سيتم إضافة العنصر لنهاية القائمة.

لاحظ أيضا أنه يتم إزاحة كل العناصر ابتداء من العنصر الواقع في الفهرس **index** لنهاية القائمة (العنصر الواقع في الموقع **last** بمقدار 1) ومن ثم إضافة العنصر الجديد في مكان الفهرس index وإضافة 1 لـ last.

يتم إلقاء استثناءات في حال محاولة إضافة عنصر لقائمة ممتلئة، وكذلك عند محاولة إدخال عنصر في فهرس خارج نطاق البيانات المخزنة.

```
public void insert(int d, int index) throws Exception
{
     if (isFull())
     {
          throw new Exception("(1): The array is full !");
     }
     else
          if((isEmpty()) || (index == (last+1)))
          {
               add(d);
          }
          else
               if((index >=0) && (index <=last))
               {
                    for(int i = last; i>=index; i--)
                    {
                         data[i+1] = data[i];
                    }
                    data[index]=d;
                    last++;
               }
               else
                    throw new Exception ("(2) : the index is out
          of boundary !");
}
```

الان يمكننا استدعاء هذا المنهج في الفصيل **UseList** كالتالي:

```
myList.insert(40, 1);
myList.insert(50, 0);
```

والذي سيؤدي لحشر العنصر 40 في الموقع 1 وحشر 50 في الموقع 0.

منهج حذف عنصر من القائمة remove

عملية حذف عنصر ضمن موقع معين من القائمة هي مشابه تماما لعملية إضافة عنصر في موقع معين بالقائمة من حيث التكلفة، حيث أن عملية حذف عنصر سيؤدي لإزاحة كل العناصر التالية له موقعا للأمام وذلك حتى لايكون هناك فراغات ضمن القائمة. المنهج سيعيد عنصر البيانات الذي تم حذفه.

لاحظ أنه سيتم انقاص 1 من الحقل last. وأن الإستثناءات يتم إلقاؤها في حال ما إن كانت القائمة فارغة أو في حال محاولة الوصول لموقع خارج نطاق البيانات المخزنة فعليا.

```
public int remove(int index) throws Exception
{
    if(isEmpty())
    {
        throw new Exception ("(3): The array is empty !");
    }
    else
            if((index >=0) && (index <= last))
            {
                int d = data[index];
                for (int i = index; i<last; i++)
                {
                    data[i] = data [i+1];
                }

                last--;
                return d;
            }
            else
                throw new Exception ("(2) : the index is out
        of boundary !");
}
```

يمكننا استدعاء هذا المنهج من الفصيل UseList كالتالي:

```
int deletedItem = mylist.remove(0);
```

هذا الاستدعاء سيقوم بحذف العنصر في الموقع 0 وإرجاع قيمته ليتم تخزينها في المتغير deletedItem.

لاحظ أيضا أنه يمكننا استدعاء هذا المنهج ضمن جملة الطباعة التالية:

```
System.out.print("Data  :  " + myList.remove(0) + "has been
removed");
```

منهج تحديث عنصر في القائمة updateItem

عملية التحديث تتم بشكل مباشر بالوصول لعنصر عن طريق فهرسه ثم تعديله. الإستثناء الوحيد الذي يتم إلقاؤه هو في حالة الوصول لفهرس خارج نطاق البيانات المخزنة.

المنهج سيكون كالتالي:

```
Public void updateItem(int index, int newltem) throws
Exception
{
      if ( (index >= 0) && (index <= last) )
       {
              data [index] = newItem;
       }
       else
              throw new Exception("(5):The index "+ index + "
    is out of boundary! ");
}
```

يتم استدعاء هذا المنهج كالتالي:

```
myList.updateItem(2, 88);
```

هنا سيتم تعديل العنصر الواقع بالموقع 2 ليأخذ القيمة الجديدة وهي: 88.

منهج استرجاع عنصر من القائمة getItem

تتم عملية استرجاع عنصر من القائمة أيضا بشكل مباشر بالوصول للعنصر عن طريق فهرسه طالما أن العنصر واقع ضمن البيانات المخزنة فعليا.

```
public int getItem(int index) throws Exception
{
     if ((index >= 0) && (index <= last))
        return data[index];
     else
        throw new Exception ("(2): the index" + index + "is
out of Boundary");
}
```

فللحصول على قيمة عنصر معين يمكننا استدعاء هذا المنهج ضمن جملة الطباعة كالتالي:

```
System.out.println(myList.getItem(2));
```

جملة الطباعة هذه ستقوم بطباعة قيمة العنصر الواقع في الموقع2.

منهج البحث عن عنصر في القائمة isFound

هذا المنهج سيقوم بالمرور على عناصر القائمة عنصرا بعنصر ليحدد فيما إذا كان عنصر معطى موجودا ضمن عناصر القائمة أو لا.

```
public boolean isFound(int element)
{
       for (int i= 0; i <= last; i++)
       {
          if (data[i] == element)
              return true;
       }
       return false;
}
```

فلتحديد ما إذا كان العنصر 33 موجودا ضمن القائمة أو لا يمكننا كتابة الإستدعاء التالي لهذا المنهج:

```
System.out.println(" Is 33 Found? " + myList.isFound(33));
```

النتيجة المرجعة ستكون إما true أو false.

منهج زيارة "طباعة" عناصر القائمة print

يمكننا المرور أو زيارة كل عنصر من عناصر القائمة، ذلك لإجراء أية عملية معالجة أخرى أو لطباعتها كالتالي:

```
public void print()
{
        for (int i = 0; i <= last; i++)
        {
           System.out.println("Data[" +i+ "] ==> " + data[i]);
        }
}
```

يتم استدعاء هذا المنهج بطريقة مباشرة لطباعة كل عناصر القائمة كالتالي:

```
myList.print();
```

التالي هو البرنامج كامل محتويا على كل المناهج المذكورة سابقا.

```
public class ListClass
{
    private int last;
    private int capacity;
    public  int[] data;

    public ListClass(int c)
    {
        capacity = c;
        data = new int[capacity];
        last = -1;
    }

    public void insert (int d, int index) throws Exception
    {
        if (isFull() )
        {
            throw new Exception("(1): The array is full !");
        }
        else
            if((isEmpty()) || (index == (last+1)))
            {
                add(d);
```

```
            }
            else
                if ((index >=0) && (index <=last))
                {
                    for (int i = last; i>=index; i--)
                    {
                        data[i+1] = data[i];
                    }
                        data[index] = d;
                        last++;
                }
                else
                    throw new Exception("(2):the index is out
of boundary!");
    }

    public void updateItem(int index, int newItem) throws
Exception
    {
        if ( (index >= 0) && (index <= last) )
        {
            data[index] = newItem;
        }
        else
            throw new Exception("(5):The index " +index+ " is
out of boundary");
    }

    public void add(int d) throws Exception
    {
        if (isFull())
        {
            throw new Exception("(1): The array is full !");
        }
        else
        {
            last++;
            data[last] = d;
        }
    }
```

```
    public int remove(int index) throws Exception
    {
        if (isEmpty())
        {
            throw new Exception("(3): The array is empty !");
        }
        else
        {
            if ((index >= 0) && (index <= last))
            {
                int d  = data[index];
                for (int i  = index; i<last; i++)
                {
                    data [i] = data [i+1];
                }

                last --;
                return d;
            }
            else
                throw new Exception ("(2): The index is out of
boundary!");

        }
    }

    public int getItem(int index) throws Exception
    {
        if ((index >= 0) && (index <= last))
            return data[index];
        else
            throw new Exception ("(2):the index " +index+ "is
out of Boundary");
    }

    public boolean isFound(int element)
    {
        for (int i = 0; i <= last; i++)
        {
            if (data[i] == element)
                return true;
        }
```

```
        return false;
    }

    public boolean isFull()
    {
        return (last == (capacity-1));
    }

    public boolean isEmpty()
    {
        return (last < 0);
    }

    public void print()
    {
        for (int i = 0; i <= last; i++)
        {
            System.out.println("Data[" +i+ "] ==> " + data[i]);
        }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل، لاحظ أننا قمنا بتعليق عدد من الجمل والتي يمكنك الإفراج عنها وتجربتها وذلك بحذف علامتي (//) ليتسنى لك رؤية تأثير كل منهج لوحده:

```
public class UseList
{
    public static void main (String[] args)
    {
        ListClass myList = new ListClass(5);

        System.out.println("\f");

        try
        {
            myList.insert(10, 0);
            myList.insert(20, 0);
            myList.insert(30, 2);
            myList.insert(40, 1);
            myList.insert(50, 0);
            myList.updateItem(2, 88);
```

```
            // myList.insert(60, 0);

            // System.out.println(myList.getItem(2));
            // System.out.println(" Is 33 Found? " + myList.isFound(33));

            // System.out.println(myList.remove(0) + "has been removed");
            // System.out.println(myList.remove(0) + "has been removed");
            // System.out.println(myList.remove(2) + "has been removed");
            // System.out.println(myList.remove(0) + "has been removed");
            // System.out.println(myList.remove(0) + "has been removed");
            // System.out.println(myList.remove(0) + "has been removed");

            myList.add(90);

        }
        catch (Exception e)
        {
            // System.out.println("ERROR" + e.getMessage());
        }

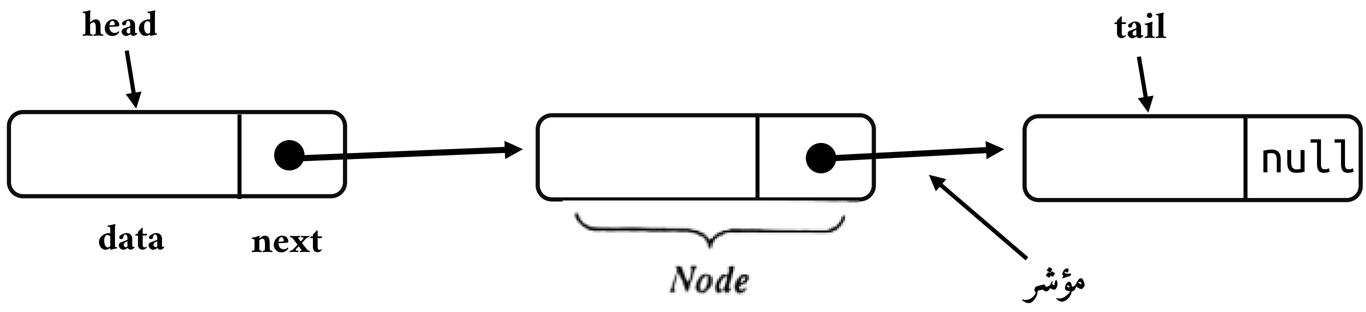
        myList.print();
    }
}
```

ثانيا تمثيل القائمة باستخدام القوائم المترابطة

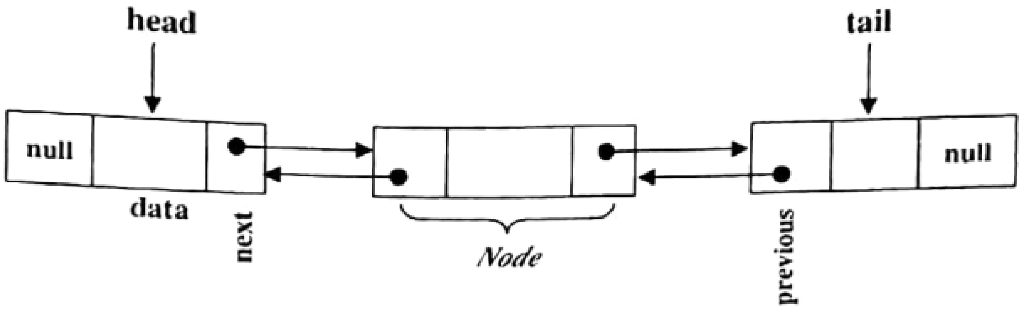
رغم أن استخدام المصفوفات لتخزين البيانات تمكننا من سرعة الوصول لعناصرها (الوصول العشوائي)، غير أن عملية إضافة أو حذف عنصر في وسط المصفوفة قد يستهلك وقتا كثيرا أثناء المعالجة؛ فبدلا من ذلك يمكننا استخدام القوائم المترابطة عندما تكون عمليتي الإضافة والحذف تشكل الأهمية الأكبر. ففي عملية الإضافة مثلا سيتم تغيير مؤشر لعقدة واحدة فقط وأن بقية عقد القائمة لا تتأثر من هذه العملية، وكذلك الحال بالنسبة للحذف فالتأثير سيكون على عقدة واحدة فقط وأما بقية القائمة فستبقى كما هي؛ غير أن عملية الوصول العشوائي (Random Access) ستكون بطيئة بعض الشيء في القوائم المترابطة، حيث سيتم المرور المتسلسل على كل العناصر السابقة للعنصر المراد الوصول إليه؛ ولكن عندما لا تكون هناك أهمية للوصول العشوائي، فيمكن استخدام القائمة المترابطة بدون أي إشكالية، مثلا: كأن يتم المرور على كل عناصر القائمة بشكل متتابع لأجل طباعتها.

القائمة المترابطة هي عبارة عن مجموعة من العقد (***Nodes***). **كل عقدة تتكون من جزئين:**

- **الجزء الأول: data**، يتم فيه تخزين البيانات (قد تكون صحيحة أو حقيقية أو سلاسل أو كائنات من فصيل معين).
- **الجزء الثاني: next**، فهو مؤشر أو مرجع للعقدة التالية في القائمة، لكل قائمة مؤشر يشير لأول عقدة يسمى رأس القائمة "head"، كما يمكن أن يحتوي على مؤشر ثانٍ يشير لأخر عقدة في القائمة يسمى ذيل القائمة "tail". أخر عقدة في القائمة لا تشير لأية عقدة تالية لها وعادة ما تأخذ القيمة Null.

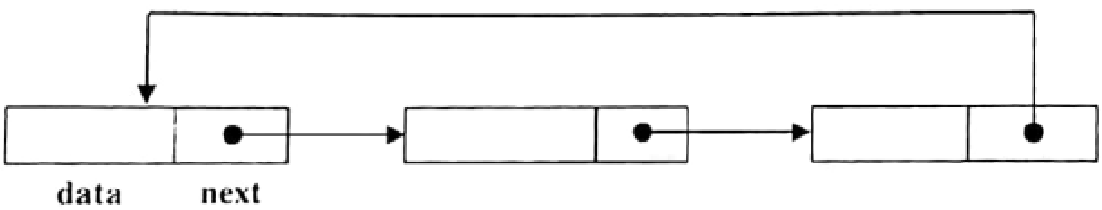


القائمة السابقة هي القائمة الخطية البسيطة، حيث أن هناك مؤشر واحد يخرج من عقدة ليشير للعقدة التالية له. ولكن العقدة قد يكون لها أكثر من مؤشر، فمثلا: في القائمة المزدوجة (***double linked list***) يكون هناك مؤشران لكل عقدة واحدة أيمن والأخر أيسر كما يلي:



القائمة المزدوجة

القائمة قد تكون دائرية بحيث أن مؤشر أخر عقدة يشير لأول عقدة في القائمة، وفي هذه الحالة يمكن اعتبار أي عقدة هي رأس القائمة، كالتالي:



القائمة الدائرية

سنقوم بتطبيق القائمة الخطية البسيطة باستخدام الجافا و إجراء مختلف عمليات المعالجة عليها من حذف وإضافة عقد وكذلك عمليات البحث و الطباعة و غيرها من أنواع المعالجة.

حيث أن القائمة المترابطة تتكون من عقد، فسنقوم أولا بتكوين فصيل للعقدة اسمه ***Node*** قبل قيامنا بتكوين فصيل للقائمة المترابطة في حد ذاتها.

كما علمنا سابقا أن كل عقدة تتكون من جزئين، هما: ***data***، وستكون من الـ *Object*، والجزء الثاني، هو: ***next***، وهو مؤشر للعقدة التالية، لذلك سيكون من نوع *Node*. سنقوم بتكوين عدة بواني بمختلف الإحتمالات لاستقبال قيم للحقلين. أيضاً سنقوم بتكوين مناهج الـ *set* والـ *get* لكل من الحقلين، لاحظ أن الحقلين سيكونا من النوع المحمي ليتم الوصول إليها مباشرة من بقية الفصائل ضمن نفس الحزمة.

```
public class Node
{
    protected Object data;
    protected Node next;

    public Node(Object item, Node ptr)
    {
        data = item;
        next = ptr;
    }
    public Node(Object item )
    {
        data = item;
        next = null;
    }
    public Node( Node ptr )
    {
        data = null;
        next = ptr;
    }
    public Node()
    {
        data = null;
        next = null;
    }
    public void setData (Object item)
    {
        data = item;
    }
    public void setNext (Node ptr)
    {
        next = ptr;
    }
    public Object getData()
    {
        return data;
    }
    public Node getNext()
    {
        return next;
    }
}
```

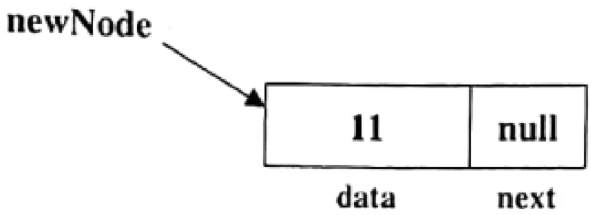
يمكننا الان استخدام هذا الفصيل لتكوين عقدة جديدة كالتالي:

```
Node newNode = new Node(11, null);
```

أو يمكن كتابتها كالتالي:

```
Node newNode = new Node(11);
```

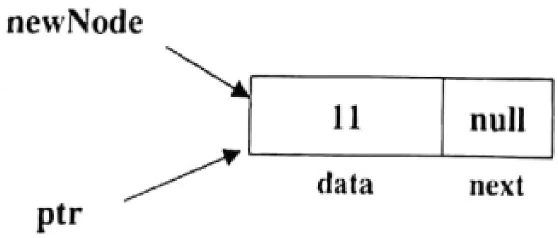
والنتيجة ستكون واحدة كما يلي:



لاحظ أنه يمكننا تخصيص المرجع *newNode* لمتغير أخر من نوع *Node*، ولكن *ptr* كالتالي:

```
Node ptr;
ptr = newNode;
```

الان ***ptr*** سيؤشر لما يؤشر له المرجع *newNode* وأن أي تغيير في أي منهما سيؤثر على الأخر، لأنهما يشيران لنفس المرجع.



فللوصول لجزء البيانات ضمن العقدة نكتب التالي:

```
int a = newNode.getData();
```

أو

```
int a = newNode.data;
```

ففي الأولى استخدمنا المنهج *getData*() للحصول على بيانات العقدة وتخزينها في المتغير الصحيح *a* باعتباره منهجا عاما يمكن الوصول له من أي فصيل أخر. أما الحالة الثانية فيمكننا استخدامها داخل فصائل نفس الحزمة أو الفصائل الوارثة باعتبار أن الـ *data* حقل محمي.

كذلك الحال بالنسبة لجزء الـ next، كالتالي:

```
Node pointer = newNode.getNext();
```

أو

```
Node pointer = newNode.next;
```

أما لتخصيص قيمة لجزء البيانات فيمكننا استخدام أي من الطريقتين التاليتين:

```
newNode.setData(55);
```

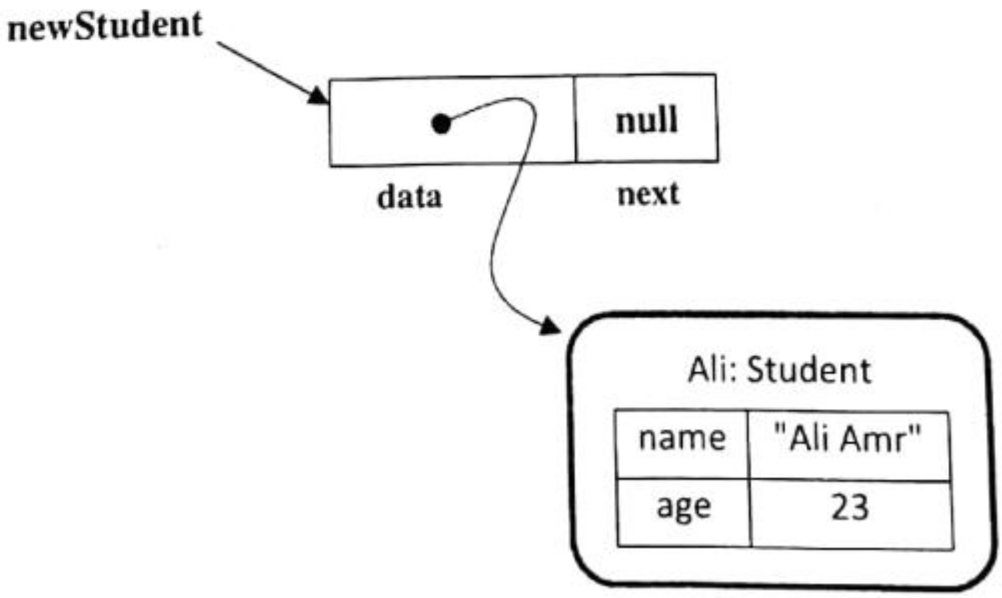
أو:

وكذلك الحال بالنسبة لـ *next*.

نوع البيانات قد يكون كائنات لفصيل أخر. فمثلا: إذا كان لدينا فصيل الطالب *Student* وبه حقلين، هما: اسم الطالب *name*، وعمره *age*، وبه مناهج: ()*getName* و ()*getAge*، يرجعان كل من الاسم والعمر على التوالي، ومنهجين أخرين، هما: ()*setName* و ()*setAge*، لتعديل حقلي الاسم والعمر. فيمكن إنشاء عقدة بيانات تحتوي على كائن الطالب، كالتالي:

```
Student ali = new Student ("Ali Amr", 23);
Node newStudent = new Node(ali);
```

يمكن تمثيل هذا بما يلي:



للوصول لاسم الطالب ضمن بيانات العقدة نكتب مايلي:

```
String stdName = newStudent.getData().getName();
```

أو

```
String stdName = newStudent.data.getName();
```

فللوصو لاسم الطالب عليك أولا بالوصول لحقل البيانات ضمن العقدة إما اسم الحقل *data* أو استدعاء المنهج *getData*()، ومن ثم يمكنك الوصول لاسم الطالب باستخدام المنهج العام *getName*() ضمن فصيل الطالب. هنا يجب ملاحظة أن البيانات *data* ضمن فصيل الـ*Node* يجب أن تكون من النوع طالب *Student*. أما في حالة استخدام بيانات من النوع *Object* فعليك أولا بعملية التحويل القسري "*Casting*" للبيانات لنوع الطالب *Student* قبل الوصول لاسم أو درجة الطالب. أي بمعنى إذا استخدمت بيانات من نوع الـ*Object* ضمن فصيل العقدة *Node* وتريد الوصول لاسم الطالب نكتب الاتي:

```
String stdName = ( (Student) newStudent.getData() . getName();
```

أو:

```
String stdName = ( (Student) newStudent.data) . getName();
```

أما لتعديل عمر الطالب فنكتب الاتي:

```
( (Student) newStudent.getData()) . setAge(18);
```

أو:

```
( (Student) newStudent.data) . setAge(18);
```

أما إذا كانت البيانات في العقدة *Node* من نوع الطالب *Student* فيمكن الوصول لاسم وعمر الطالب مباشرة بدون إجراء التحويل القسري "*Casting*" وكالتالي:

```
newStudent.getData().setName("Nuri Sami");
```

أو:

```
newStudent.data.setName("Nuri Sami");
```

قبل القيام بإجراء العمليات المختلفة على القائمة المترابطة، سنقوم أولا بتكوين فصيل جديد اسمه (LinkedList) يحتوي على حقلين، أحدهما مؤشر لأول عقدة في القائمة "head" والثاني مؤشر لأخر عقدة في القائمة "tail"، كالتالي:

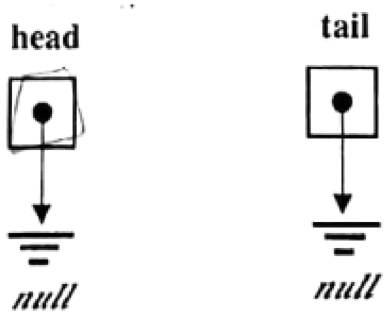
```
public class LinkedList
{
     protected Node head;
     protected Node tail;

     public LinkedList()
     {
          head = null;
          tail = null;
     }
}
```

الان يمكننا استخدام هذا الفصيل ضمن فصيل أخر، وليكن (*UseLinkedList*) وتكوين كائن من القائمة المترابطة كالتالي:

```
LinkedList myList = new LinkedList();
```

لاحظ أنه عند تكوين كائن جديد من فصيل القائمة المترابطة فإن القائمة ستكون فارغة وأن كل من مؤشر الرأس والذيل يكون مساويا لـ *null* كما بالشكل التالي:



إضافة عقدة للقائمة المترابطة

عملية إضافة عقدة للقائمة المترابطة قد تتم عند رأس القائمة أو ذيل القائمة أو إضافة عقدة قبل عقدة معينة أو بعدها. سنقوم بشرح كل عملية لوحدها.

منهج إضافة عقدة لنهاية القائمة المترابطة addToEnd

تعتبر عملية الإضافة هذه من أبسط أنواع الإضافات، حيث يتم إضافة عقدة جديدة عند النهاية بحيث يخزن جزء البيانات القيمة الجديدة المراد إضافتها ووضع null لجزء الـ next، لأن هذه العقدة ستكون أخر عقدة في القائمة، ثم نغيير مؤشر التالي "next" للعقدة الأخيرة (عقدة الـ tail) ليؤشر للعقدة الجديدة. فقط هناك حالة خاصة يجب أخذها في الإعتبار وهي عندما تكون القائمة فارغة فإن مؤشر عقدة الرأس سيؤشر أيضا على العقدة الجديدة كونها العقدة الوحيدة في القائمة المترابطة، كما بالمنهج التالي:

```
public void addToEnd (Object item)
{
    Node newNode  = new Node (item, null);

    if (isEmpty())
        head  = newNode;
    else
        tail.next = newNode;

    tail = newNode;
}
```

منهج إضافة عقدة عند رأس القائمة المترابطة addAtFirst

تتم هذه العملية بإضافة عقدة جديدة تحمل القيمة المراد إضافتها عند بداية القائمة المترابطة، ومؤشر العقدة الجديدة سيؤشر إلى ما يؤشر إليه رأس القائمة مع تغيير مؤشر رأس القائمة ليؤشر إلى هذه العقدة الجديدة، الحالة الخاصة هنا هي عندما تكون القائمة فارغة، فإن مؤشر الذيل سيؤشر على هذه العقدة الجديدة أيضا، كما هو موضح في المنهج التالي:

```
public void addAtFirst (Object item)
{
    Node newNode = new Node(item, head);

    if (isEmpty())
        tail = newNode;

    head = newNode;
}
```

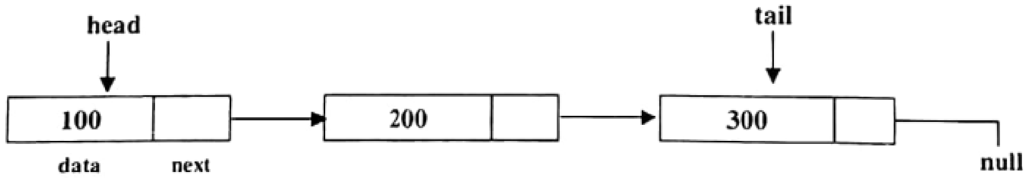
لاحظ أننا استخدمنا المنهج *isEmpty*() لاختبار ما إذا كانت القائمة المترابطة فارغة أم لا. تكون القائمة المترابطة فارغة عندما يكون أي من مؤشري الرأس أو الذيل مساويا لـ *null* فيمكن برمجة هذا المنهج كالتالي:

```
public boolean isEmpty()
{
    return (head == null);
}
```

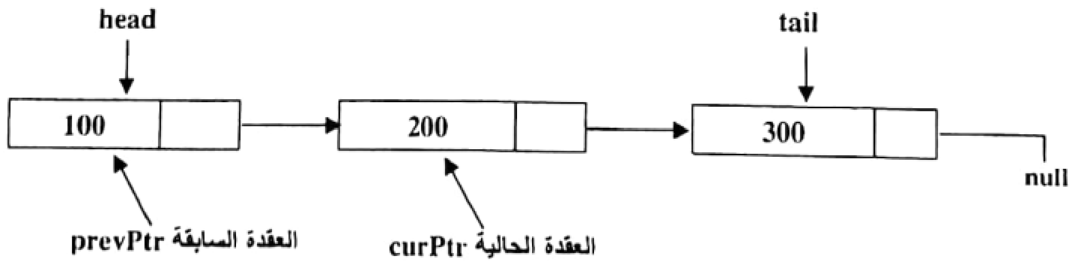
منهج إضافة عقدة قبل عقدة معينة في القائمة المترابطة insertBefore

تتم هذه العملية بإيجاد مؤشرين واحد للعقدة المراد الإضافة قبلها ***curPtr*** والمؤشر الأخر للعقدة السابقة للعقدة المراد الإضافة قبلها ***prevPtr***. ثم نجعل مؤشر الـ ***next*** للعقدة الجديدة يشير للعقدة المراد الإضافة قبلها (أي مساويا لـ ***curPtr***) ومؤشر الـ ***next*** للعقدة السابقة (***prevPtr***) يشير للعقدة الجديدة. والمثال التالي يوضح هذه العملية:

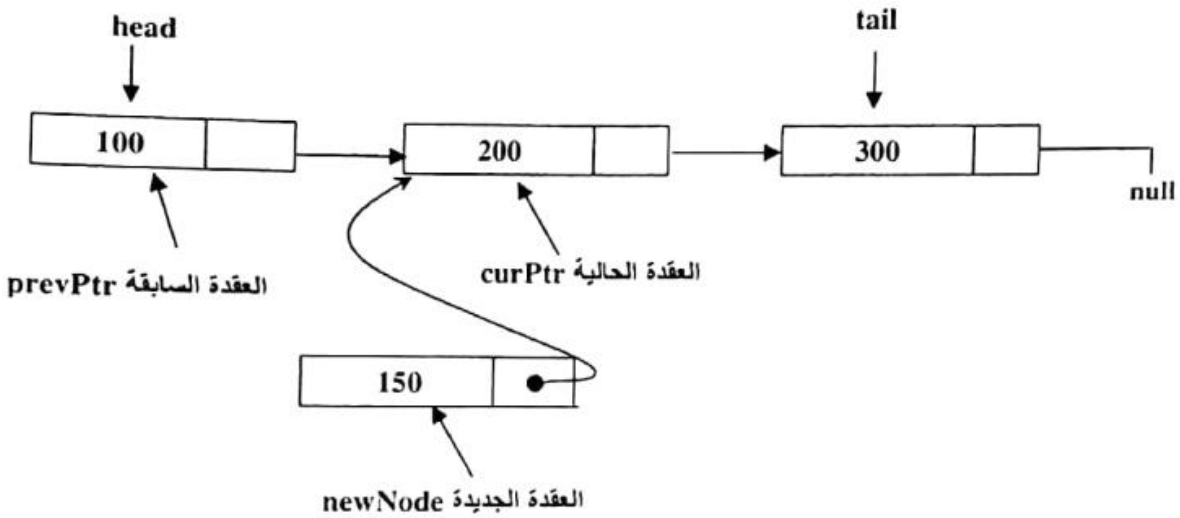
لنفترض أن لدينا القائمة التالية ونريد إضافة عقدة جديدة قبل الـ(200).



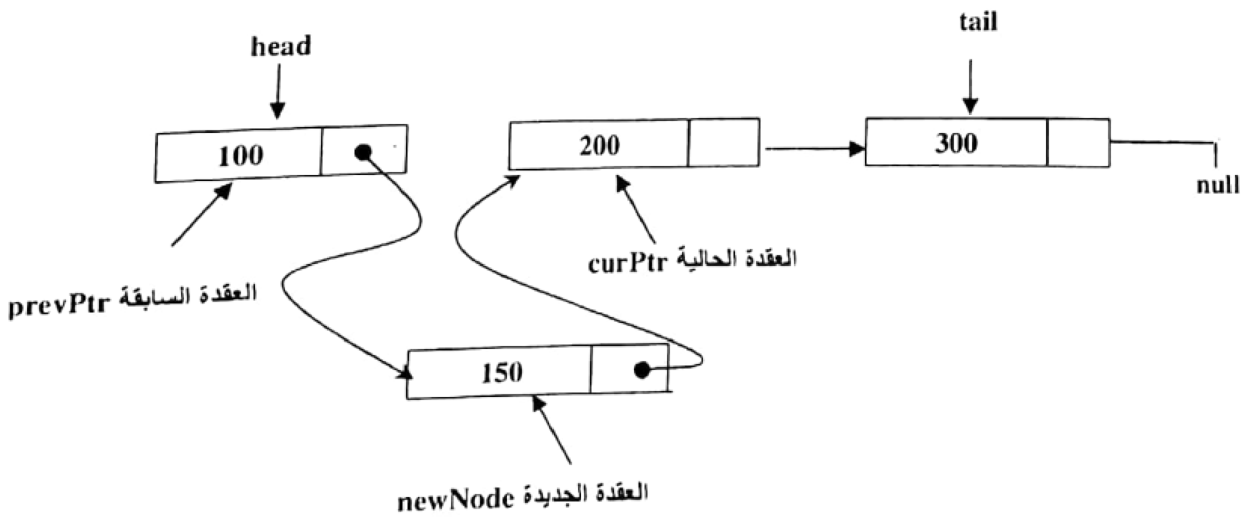
1- نبحث عن العقدة (200) ونحدد موقعها (***curPtr***) وموقع العقدة السابقة لها (***prevPtr***).



2- ننشئ عقدة جديدة (newNode) وندخل بيانات العقدة الجديدة، وكذلك المؤشر (حيث يشير مؤشر الـ next للعقدة الجديدة إلى العقدة الحالية (curPtr)).



3- نحدث مؤشر الـ next للعقدة السابقة لتشير إلى العقدة الجديدة (newNode).



وبهذا نكون قد أضفنا عقدة جديدة قبل عقدة أخرى، وأن بقية العقد في القائمة لم تتأثر (أي لم تتحرك) بهذه الإضافة، وهذا ما يجعل عملية الإضافة (والحذف أيضا) في القوائم المترابطة أكثر فاعلية مقارنة بما هو موجود في المصفوفات.

لاحظ أنه لكي تتم عملية الإضافة قبل عقدة معينة يجب علينا كتابة منهج لإيجاد مؤشر يشير إلى العقدة الحالية * والمراد الإضافة قبلها، وهو كالتالي:

```
public Node currentpointer(Object obj)
{
    Node curPtr = getHead();
    while (curPtr != null)
    {
        if (curPtr.getData().equals(obj))
            return curPtr;
        curPtr = curPtr.next;
    }
      return null;
}
```

هذا المنهج يستقبل البيانات المراد إيجاد مؤشرا لها كمعلمة *obj*، ثم يمر على كل عقد القائمة واحدة تلو الأخرى بشكل تتابعي ابتداءً من عقدة الرأس إلى نهاية القائمة، وفي أثناء المرور على العقد يختبر قيمة بيانات كل عقدة ومقارنتها مع الـ *obj*، في حال حدوث تطابق بينهما يقوم بإرجاع مؤشر لتلك العقدة، أما في حالة الوصول لنهاية القائمة مع عدم الحصول على حالة تطابق يقوم المنهج بإرجاع قيمة *null*.

أيضا لكي تتم عملية الإضافة قبل عقدة معينة يجب علينا كتابة منهج أخر لإيجاد مؤشر يشير إلى العقدة السابقة ***prevPtr*** للعقدة المراد الإضافة قبلها، وهو كالتالي:

```
public Node previousPointer(Node curPtr)
{
    Node prevPtr = head;
    while ( (prevPtr != null) && (prevPtr.next != curPtr))
        prevPtr = prevPtr.next;
    return prevPtr;
}
```

هنا المنهج سيقوم باستقبال مؤشر للعقدة المراد إيجاد مؤشر للعقدة السابقة لها. ويقوم بالمرور على كل العقد في القائمة الواحدة تلو الأخرى إلى أن يصل إلى المؤشر المطلوب فيرجعه أو يرجع *null* في حالة عدم وجوده.

في الحقيقة أن المنهجين السابقين مهمان جدا لتسهيل إجراء بقية العمليات على القوائم المترابطة من إضافة وحذف عقد. المنهج التالي يوضح كيفية استخدام هذين المنهجين في عملية إضافة عقدة ما قبل عقدة معينة ضمن القائمة المترابطة:

```
public void insertBefore(Object currentData, Object dataToAdd)
throws Exception
{
    Node curPtr = currentpointer(currentData);

    if(curPtr == null)
        throw new Exception(currentData + " is not found!");
    else
    {
        Node newNode = new Node(dataToAdd, curPtr);

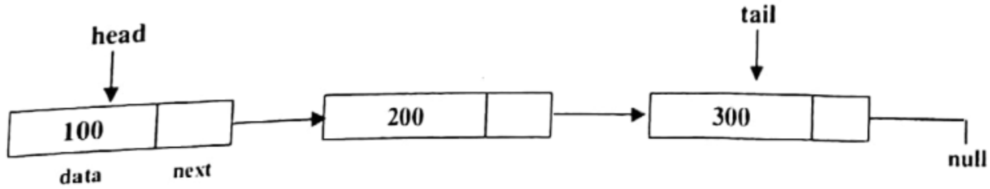
        if (curPtr == head)
            head = newNode;
        else
        {
            Node prevPtr = previousPointer(curPtr);
            prevPtr.next = newNode;
        }
    }
}
```

لاحظ أنه عند محاولة الإضافة إلى رأس القائمة فسيتم تغيير مؤشر الرأس إلى العقدة الجديدة. أما الإستثناء فسيتم حذفه في حالة ما إن كان عنصر البيانات المراد الإضافة قبله ليس موجود أصلا ضمن القائمة.

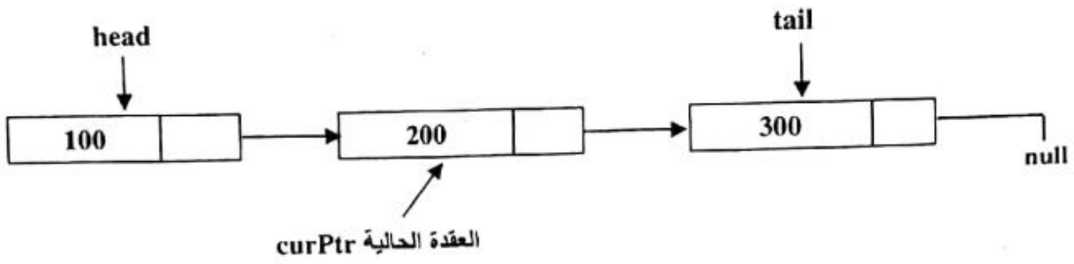
منهج إضافة عقدة بعد عقدة معينة في القائمة المترابطة insertAfter

تتم هذه العملية بإيجاد مؤشر للعقدة المراد الإضافة بعدها curPtr. ثم نجعل مؤشر الـ next للعقدة الجديدة يشير لما يؤشر إليه مؤشر الـ next للعقدة المراد الإضافة بعدها (أي مساويا لـ curPtr.next) ومؤشر الـ next للعقدة المراد الإضافة بعدها (curPtr) يشير إلى العقدة الجديدة. والمثال التالي يوضح هذه العملية:

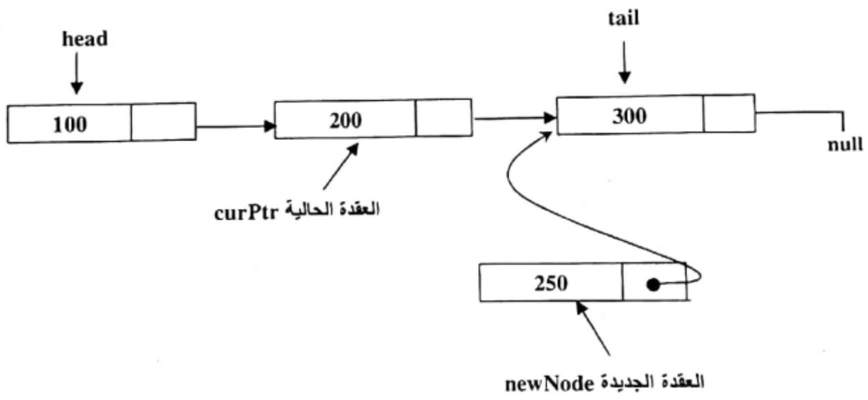
لنفترض أن لدينا القائمة التالية ونريد إضافة عقدة جديدة بعد الـ(200).



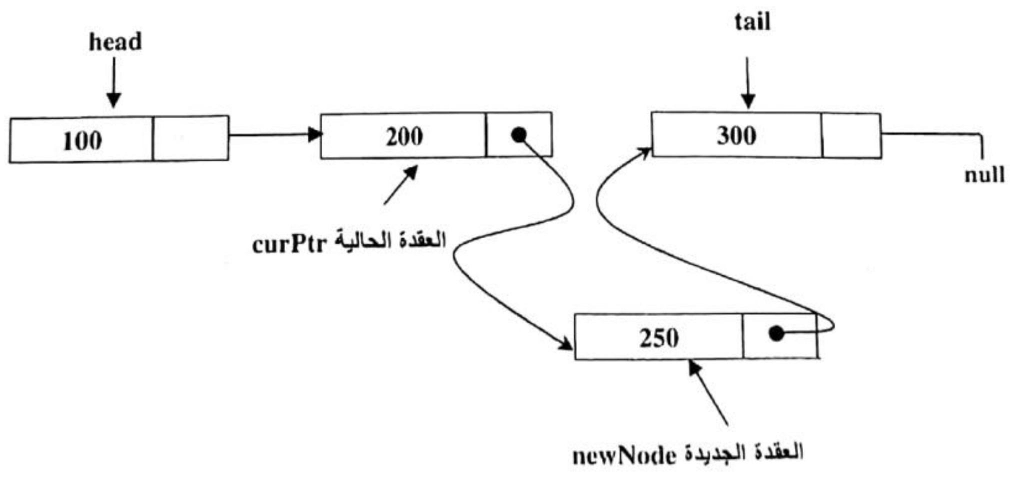
1- نبحث عن العقدة (200) ونحدد موقعها (curPtr)



2- ننشئ عقدة جديدة (newNode) وندخل بيانات العقدة الجديدة، وكذلك المؤشر (حيث يشير مؤشر العقدة الجديدة إلى ما يشير إليه جزء الـ next للعقدة الحالية (curPtr)).



3- نحدث مؤشر الـ next للعقدة الحالية لتشير إلى العقدة الجديدة (newNode).



والمنهج لإجراء هذه العملية هو كالتالي:

```
public void insertAfter(Object currentData, Object dataToAdd)
throws Exception
{
    Node curPtr = currentpointer(currentData);

    if (curPtr == null)
        throw new Exception (currentData +  " is not found");
    else
    {
        Node newNode = new Node (dataToAdd, curPtr.next);

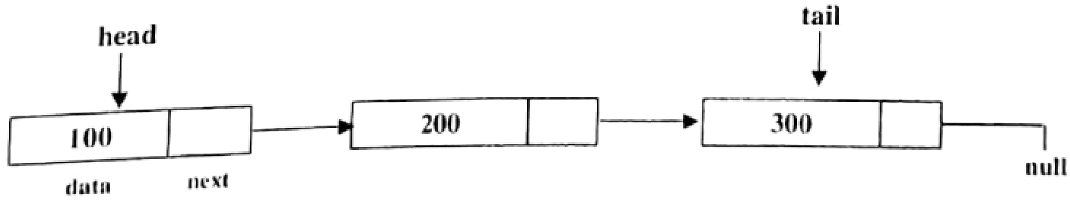
        curPtr.next=newNode;
        if (curPtr == tail)
            tail = curPtr.next;
    }
}
```

لاحظ أنه عند محاولة إضافة عقدة بعد أخر عقدة في القائمة أي بعد الـ ***tail*** فسيتم تغيير مؤشر الذيل ***tail*** ليشير إلى العقدة الجديدة.

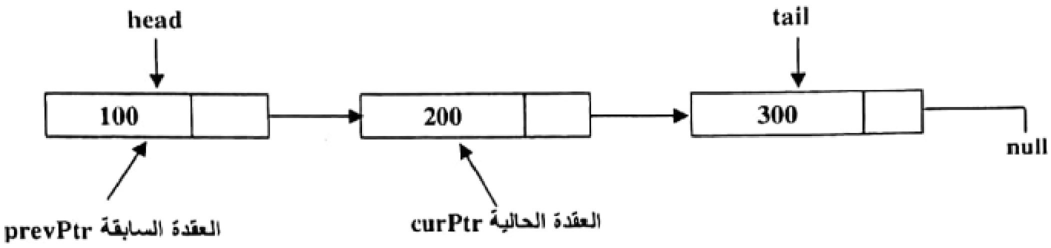
حذف عقدة من القائمة المترابطة remove

عملية حذف عقدة من القائمة المترابطة قد تتم عند رأس القائمة أو ذيل القائمة أو وسط القائمة المترابطة. على كل حال سنقوم بتكوين منهج واحد يقوم بعملية الحذف من أية جهة. تتم عملية حذف عقدة ما ضمن القائمة بإيجاد مؤشرين واحد للعقدة المراد حذفها ***curPtr***، والمؤشر الآخر للعقدة السابقة للعقدة المراد حذفها prevPtr. ثم نجعل مؤشر الـ next للعقدة السابقة يشير إلى ما يؤشر إليه مؤشر الـ ***next*** للعقدة الحالية. والمثال التالي يوضح هذه العملية:

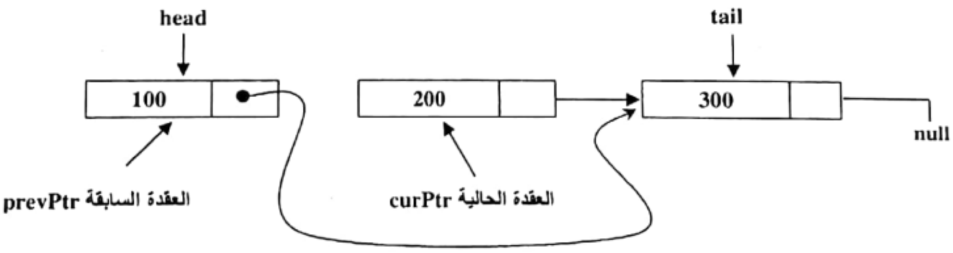
لنفترض أن لدينا القائمة التالية ونريد حذف العقدة (200).



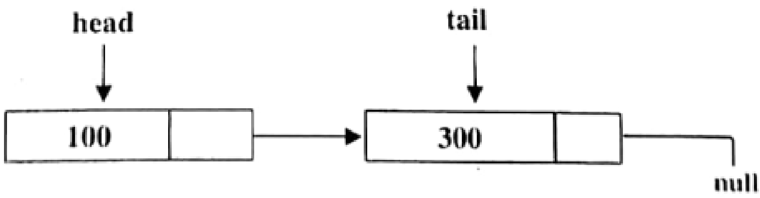
1- نبحث عن العقدة (200) ونحدد موقعها (curPtr) وموقع العقدة السابقة لها (prevPtr).



2- نغير مؤشر الـ next للعقدة السابقة (أي pervPtr) ليشير إلى ما يشير إليه الـ next للعقدة الحالية (أي curPtr.next) كالتالي:



وبالتالي لن يكون هناك مؤشر إلى العقدة القديمة curPtr وأن المنظف في الجافا "*garbage collector*" سوف يتخلص منها ويعيدها للذاكرة مرة أخرى نيابة عنا، فلا داعي للتخلص منها يدويا كما هو الحال في لغة السي مثلا، وأن القائمة ستصبح:



والمنهج التالي يقوم بعملية حذف عقدة من أية جهة ضمن القائمة المترابطة:

```
public void remove (Object element) throws Exception
{
    Node curPtr = currentpointer(element);

    Node prevPtr= previousPointer(curPtr);

    if (curPtr == null)
        throw new Exception (element + " is not found");
```

```
    if (curPtr == head)
        head = curPtr.next;
    else
        prevPtr.next = curPtr.next;

    if (curPtr == tail)
        tail = prevPtr;
}
```

لاحظ أن المنهج يستقبل قيمة عنصر البيانات المراد حذفه ومن ثم يوجد مؤشر له ضمن القائمة curPtr وبعد ذلك يقوم بإيجاد مؤشر للعقد السابقة prevPtr للعقدة الحالية. يجب التنويه إلى أنه عند محاولة حذف عقدة رأس القائمة فإن مؤشر الرأس سوف يشير بعد ذلك إلى العقدة التالية له. وكذلك الحال بالنسبة لحذف عقدة ذيل القائمة فمؤشر الذيل سيشير إلى العقدة السابق له. يتم إلقاء الإستثناء عند محاولة حذف عنصر بيانات ليس موجود ضمن القائمة المترابطة.

منهج زيارة أو طباعة عناصر القائمة المترابطة printLinkedList

يمكننا المرور أو زيارة كل عقدة من عقد القائمة المترابطة ابتداء من عقد الرأس وحتى الوصول إلى ذيلها وذلك لإجراء أية عملية معالجة أخرى أو لطباعتها كالتالي:

```
public void printLinkedList()
{
    for (Node p = head; p != null; p = p.next)
    {
        System.out.println(p.data);
    }
}
```

التالي هو البرنامج كاملا محتويا على كل المناهج سالفة الذكر.

```
public class LinkedList
{
    protected Node head;
    protected Node tail;

    public LinkedList()
    {
        head = null;
     tail = null;
    }

    public void addToEnd (Object item)
    {
        Node newNode  = new Node (item, null);

        if (isEmpty())
            head  = newNode;
        else
            tail.next = newNode;
        tail = newNode;
    }

    public void addAtFirst(Object item)
    {
        Node newNode = new Node(item, head);

            if (isEmpty())
                tail = newNode;

            head = newNode;
    }

    public boolean isEmpty()
    {
        return (head == null);
    }

    public Node getHead()
    {
        return head;
    }
```

```
    public Node getTail()
    {
        return tail;
    }

    public void printLinkedList()
    {
        for (Node p = head; p != null; p = p.next)
        {
            System.out.println(p.data);
        }
    }

    public Node currentpointer(Object obj)
    {
        Node curPtr = getHead();
            while (curPtr != null)
            {
                if (curPtr.getData().equals(obj))
                    return curPtr;
                curPtr = curPtr.next;
            }
            return null;
    }

    public Node previousPointer(Node curPtr)
    {
        Node prevPtr = head;

        while ( (prevPtr != null) && (prevPtr.next != curPtr))
               prevPtr = prevPtr.next;

        return prevPtr;
    }

    public   void   insertBefore(Object   currentData,   Object
dataToAdd) throws Exception
    {
        Node curPtr = currentpointer(currentData);
            if(curPtr == null)
                throw  new  Exception(currentData  +  "  is  not
found!");
```

```
            else
            {
                Node newNode = new Node(dataToAdd, curPtr);
                if (curPtr == head)
                    head = newNode;
                else
                {
                    Node prevPtr = previousPointer(curPtr);
                    prevPtr.next = newNode;
                }
            }
    }
```

public void insertAfter(Object currentData, Object dataToAdd) throws Exception

```
    {
        Node curPtr = currentpointer(currentData);
            if (curPtr == null)
                throw new Exception (currentData +  " is not found");
            else
            {
                Node newNode = new Node(dataToAdd, curPtr.next);
                curPtr.next=newNode;
                if (curPtr == tail)
                    tail = curPtr.next;
            }
    }
```

public void remove (Object element) throws Exception

```
    {
        Node curPtr = currentpointer(element);

        Node prevPtr= previousPointer(curPtr);

            if (curPtr == null)
                throw new Exception (element + " is not found");

            if (curPtr == head)
                head = curPtr.next;
            else
```

```
                prevPtr.next = curPtr.next;

            if (curPtr == tail)
                tail = prevPtr;
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل:

```
public class UseLinkedList
{
    public static void main(String[] args)
    {
        try
        {
            LinkedList myList = new LinkedList();

            myList.addToEnd(10);
            myList.addToEnd(20);
            myList.addToEnd(30);
            myList.addAtFirst(40);
            myList.addAtFirst(50);
            myList.insertBefore(30, 99);
            myList.insertAfter(10, 88);
            myList.remove(10);
            System.out.print("\f");
            myList.printLinkedList();
        }
        catch (Exception e)
        {
            System.out.print(e.getMessage());
        }
    }
}
```

المكـدس Stack

المكدس: هو هيكل من هياكل البيانات الخطية؛ فهو يمثل مجموعة من العناصر المنظمة في تسلسل. قد تكون هذه العناصر أعداد صحيحة أو حقيقية أو سلاسل حرفية أو كائنات لأصناف معينة. يعتبر المكدس نوع خاص من القائمة حيث أن عمليتي الإضافة والحذف تتم من جهة واحدة تسمى: رأس القائمة أو القمة (Top)، يعمل المكدس بمبدأ الداخل أخير أو الخارج أولا (Last In First Out - LIFO) أي أن أخر عنصر يتم إضافته سيكون أو عنصر يتم حذفه. عادة ما يتم تمثيل المكدس باستخدام المصفوفات (arrays) أو القوائم المترابطة (Linked list).

سنقوم بتمثيل المكدس باستخدام الطريقتين لإجراء المعالجة المختلفة منها على سبيل المثال:

- ❖ إضافة عنصر للمكدس وتعرف بعملية الدفع (push).
- ❖ حذف عنصر من المكدس وتعرف بعملية السحب (pop).
- ❖ البحث عن عنصر في المكدس واسترجاعه.
- ❖ زيارة عناصر المكدس بشكل تسلسلي (لأجل الطباعة مثلا).

أولا: تمثيل المكدس باستخدام المصفوفات

نظرا لأن عمليتي الإضافة والحذف تتم عند قمة المكدس أي عند أخر عنصر تم إضافته فإن استخدام المصفوفات لتمثيل المكدس يعتبر مجدي إلى حد ما، وهذا على عكس القائمة؛ حيث لا يوجد عمليات حذف أو إضافة عند منتصف أو بداية المكدس. غير أنه قد يتم حجز مساحة من الذاكرة مسبقا للمصفوفة والتي قد لا يتم استغلالها بشكل فعال في المكدس.

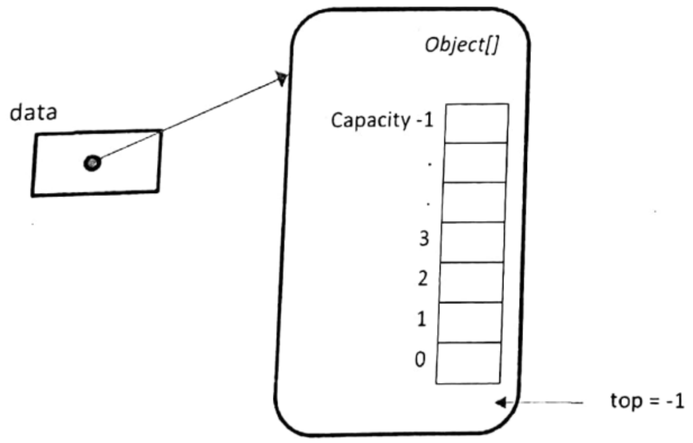
ونظرا لكون عمليتي الإضافة والحذف تتم عند القمة فسنستخدم مؤشر اسمه ***top*** يدل على موقع أخر عنصر بالمكدس ويزداد بمقدار 1 مع كل عملية إضافة، وينقص بمقدار 1 مع كل عملية حذف.

قبل القيام بإجراء العمليات المختلفة على المكدس باستخدام المصفوفات سنقوم أولا بتكوين فصيل جديد اسمه ***StackClass*** يحتوي على تعريف لمصفوفة من الكائنات (هنا سنستخدم الأعداد الصحيحة) اسمها ***data*** ذات حجم أو سعة (***capacity***) يتم تحديدها ضمن بانيها بالإضافة لحقل اسمه ***top*** يدل على موقع أخر عنصر بالمكدس كما يلي:

```
public class StackClass
{
    private int top;
    private int capacity;
    public Object[] data;

    public StackClass(int c)
    {
        capacity = c;
        data = new Object[capacity];
        top = -1;
    }
}
```

لاحظ أن حجم المكدس ***capacity*** يتم استقباله كمعلمة ضمن الباني، كذلك عند بناء المكدس لأول مرة فإن الدليل top سيكون عند 1- وذلك لعدم وجود عناصر بالمكدس، هذا الدليل سيزداد بإضافة عناصر للمكدس وينقص مع عمليات حذف العناصر، فهو يدل على عدد البيانات الفعلية المخزنة بالمكدس على عكس المتغير ***capacity*** والذي يدل على الحجم الإجمالي للعناصر التي يمكن أن يسعها المكدس. هذا التعريف سيؤدي إلى تكوين مكدس (مصفوفة) فارغ من العناصر كالتالي:

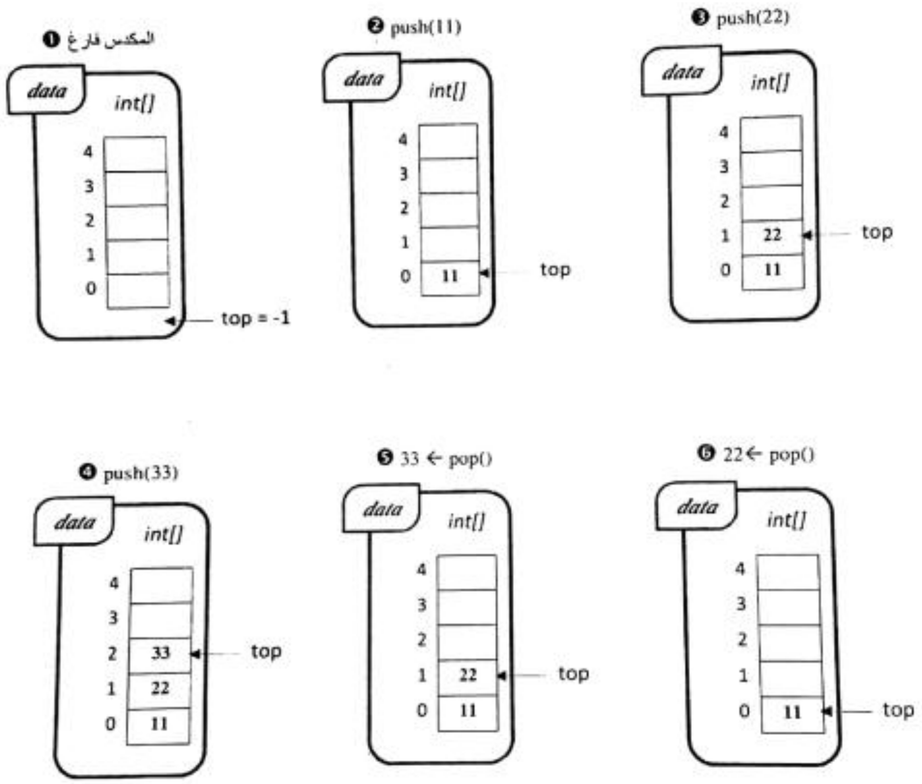


الان يمكننا استخدام هذا الفصيل ضمن فصيل آخر وليكن UseStack وتكوين كائن من المكدس كالتالي:

```
StackClass myStack = new StackClass(5);
```

هنا حددنا سعة المكدس بـ 5 عناصر كحد أقصى. الرقم 5 سيمرر إلى باني الفصيل ***StackClass*** ليتم تخزينه ضمن الحقل ***capacity*** ثم تحديد طول المكدس بـ5.

قبل القيام ببرمجة مناهج الإضافة والحذف سنقوم بتوضيح عمليتي الـ push والـ pop كما بالمخطط التالي:



منهج الإضافة للمكدس push

الان سنقوم بإضافة منهج جديد اسمه push() والذي سيقوم بإضافة عنصر جديد يتم استقباله كمعلمة لقمة المكدس. في حال ما إن كان المكدس ممتلئ فسيتم إلقاء استثناء بذلك. مع كل عملية إضافة جديدة سيتم زيادة 1 لقيمة الحقل top.

يمكننا استدعاء هذا المنهج الان ضمن الفصيل UseStack عدة مرات وكالتالي:

```
myStack.push(11);
myStack.push(22);
public void push (Object d) throws Exception
{
     if(isFull() == true)
          throw new Exception("(1): The Stack is full !");
    else
     {
          top++;
           data[top] = d;
     }
}
```

منهج هل المكدس ممتلئ isFull

لاحظنا من المنهج السابق أننا استخدمنا المنهج ()isFull ليختبر ما إن كان المكدس ممتلئ أم لا. شرط الإمتلاء يتحقق عندما تكون قيمة top مساوية لـ4 أي حجم المكدس الكلي ناقصا واحد (capacity-1)، فالمنهج كالتالي:

```
public boolean isFull()
{
     return (top == (capacity-1));
}
```

منهج هل المكدس فارغ isEmpty

يكون المكدس فارغا عندما يكون مؤشر الـtop أقل من صفر، فالمنهج يكون كالتالي:

```
public boolean isEmpty()
{
     return (top < 0);
}
```

منهج حذف عنصر من المكدس pop

عملية الحذف تتم لأخر عنصر تم إضافته أي العنصر الموجود عند القمة "top". هذا المنهج سيعيد عنصر البيانات الذي تم حذفه. لاحظ أنه سيتم إنقاص 1 من top، وأن استثناءً يتم إلقاؤه في حال ما إن كان المكدس فارغا.

يمكنك استدعاء هذا المنهج من الفصيل UseStack كالتالي:

```
System.out.println(myStack.pop() + " at the top has been removed");
```

```
public Object pop() throws Exception
{
     if(isEmpty())
          throw new Exception (" (2) : The Stack is empty !");
    else
     {
          Object d = data [top];
           top--;
           return d;
```

```
    }
}
```

منهج البحث عن عنصر في المكدس isFound

هذا المنهج سيقوم بالمرور على عناصر المكدس عنصرا بعنصر ابتداءً من أول عنصر ووصولا إلى العنصر الموجود عند القمة ***top*** ليحدد فيما إذا كان العنصر المعطى موجودا ضمن عناصر المكدس أو لا.

```
public boolean isFound (Object element)
{
    for (int i = 0; i <= top; i++)
    {
        if (data[i].equals(element))
            return true;
    }
    return false;
}
```

فمثلا لتحديد ما إذا كان العنصر 22 موجودا ضمن المكدس أو لا يمكننا كتابة الإستدعاء التالي لهذا المنهج:

```
System.out.println("Is 22 Found? " + myStack.isFound(22));
```

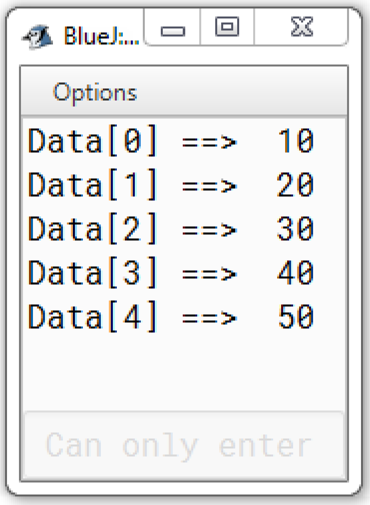
منهج زيارة "طباعة" عناصر المكدس print

يمكننا المرور أو زيارة كل عنصر من عناصر المكدس وذلك لإجراء أية عملية معالجة أخرى أو لطباعته كالتالي:

```
public void print()
{
    for (int i = 0; i <= top; i++)
    {
        System.out.println("Data[" +i+ "] ==> " + data[i]);
    }
}
```

عند استدعاء منهج الطباعة السابق فالنتيجة ستكون كما بالشكل التالي:

```
myStack.print();
```

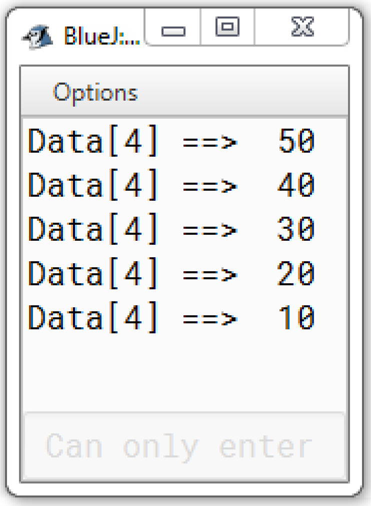


أما لطباعة المكدس بشكل عكسي ليكون منطقيا أكثر (أي يكون أخر عنصر مضافا عند القمة في الأعلى ف) نكتب المنهج التالي:

```
public void reversePrint()
{
     for (int i = top; i >= 0; i--)
         {
             System.out.println("Data[" +i+ "] ==> " + data[i]);
         }
}
```

يمكننا استدعاء هذا المنهج كما يلي والنتيجة ستكون كالتالي:

```
myStack.reversePrint();
```



التالي هو البرنامج كاملا محتويا على كل المناهج السابقة.

```
public class StackClass
{
    private int top;
    private int capacity;
    public Object[] data;

    public StackClass(int c)
    {
        capacity = c;
        data = new Object[capacity];
        top = -1;
    }

    public void push (Object d) throws Exception
    {
        if(isFull() == true)
            throw new Exception("(1): The Stack is full !");
        else
        {
            top++;
            data[top] = d;
        }
    }

    public Object pop() throws Exception
    {
        if(isEmpty())
            throw new Exception (" (2) : The Stack is empty !");
        else
        {
            Object d = data [top];
            top--;
            return d;
        }
    }

    public boolean isFound (Object element)
    {
        for (int i = 0; i <= top; i++)
        {
            if (data[i].equals(element))
                return true;
```

```
        }
        return false;
    }

    public boolean isFull()
    {
        return (top == (capacity-1));
    }

        public boolean isEmpty()
    {
        return (top < 0);
    }

    public void print()
    {
        for (int i = 0; i <= top; i++)
        {
            System.out.println("Data["+i+"] ==> " + data[i]);
        }
    }

    public void reversePrint()
    {
        for (int i = top; i >= 0; i--)
        {
            System.out.println("Data[" +i+ "] ==> " + data[i]);
        }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل، لاحظ أننا هنا أيضا قمنا بتعليق عدد من الجمل والتي يمكنك الإفراج عنها وتجربتها وذلك بحذف علامتي التعليق (//):

```
public class UseStack
{
    public static void main(String[] args)
    {
        StackClass myStack = new StackClass(5);

        System.out.println("\f");
```

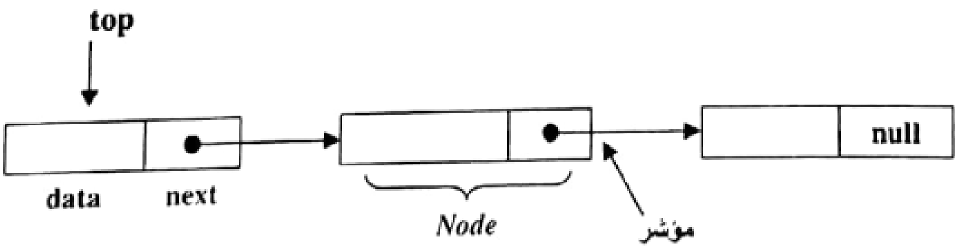
```
        try
        {
            myStack.push(11);
            myStack.push(22);
            myStack.push(33);
            myStack.push(44);
            myStack.push(55);
            // myStack.push(66);
            // System.out.println(myStack.pop() + " at the top
has been removed");
            // System.out.println(myStack.pop() + " at the top
has been removed");

            //    System.out.println("Is    22    Found?    "    +
myStack.isFound(22));
        }
        catch (Exception e)
        {
            System.out.println("ERROR" + e.getMessage());
        }

        //myStack.print();
        myStack.reversePrint();
    }
}
```

ثانيا تمثيل المكدس باستخدام القوائم المترابطة

المكدس باستخدام القائمة المترابطة: هو عبارة عن مجموعة من العقد، كل عقدة تتكون من جزئين، الجزء الأول: ***date*** يتم فيه تخزين البيانات، والجزء الثاني: ***next*** هو مؤشر أو مرجع للعقدة التالية في المكدس. هناك أيضا مؤشر يشير لقمة المكدس ***top*** (أخر عقدة تم إدخالها وأول عقدة سيتم حذفها).



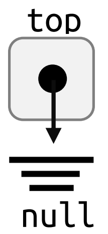
لاحظ أن فصيل العقدة الذي سنستخدمه في المكدس هو نفس فصيل العقدة الذي استخدمناه في القائمة المترابطة لذلك لن نعيد برمجته مرة أخرى. قبل القيام بإجراء العمليات المختلفة على القائمة المترابطة (المكدس)، سنقوم أولا بتكوين فصيل جديد أسمه ***Stack Class*** يحتوي على حقل واحد وهو مؤشر لقمة المكدس ***top*** كما يلي:

```
public class StackClass
 {
    protected Node top;
    public StackClass()
    {
        top = null;
    }
}
```

الان يمكننا استخدام هذا الفصيل ضمن فصيل أخر وليكن UseStack وتكوين كائن من المكدس كالتالي:

```
 StackClass stack = new StackClass( );
```

لاحظ أنه عند تكوين كائن جديد من فصيل المكدس فإن المكدس سيكون فارغ وأن مؤشر القمة top يكون مساويا للnull، كما بالشكل التالي:



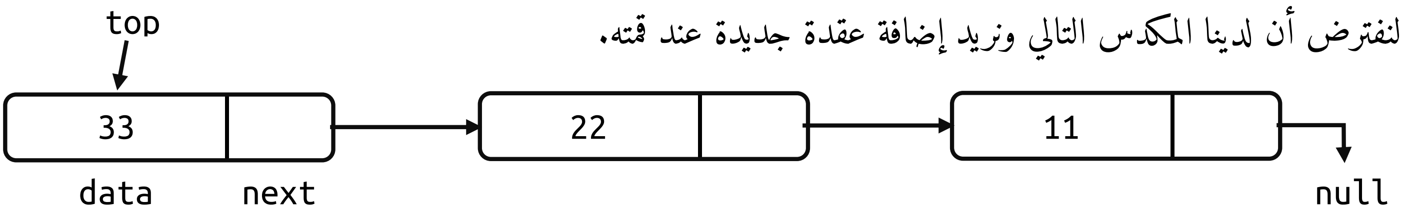
منهج إضافة عقدة للمكدس push

هذ المنهج يقوم بإضافة عقدة جديدة عند قمة المكدس بحيث يخزن جزء البيانات (data) القيمة الجديدة المراد إضافتها، وتخصيص ما يؤشر إليه مؤشر إليه top لجزء الـ next. ثم نغيير مؤشر القمة (top) للمكدس ليؤشر إلى العقدة الجديدة كونها العقدة التي ستكون في قمة المكدس، وكما بالمنهج التالي:

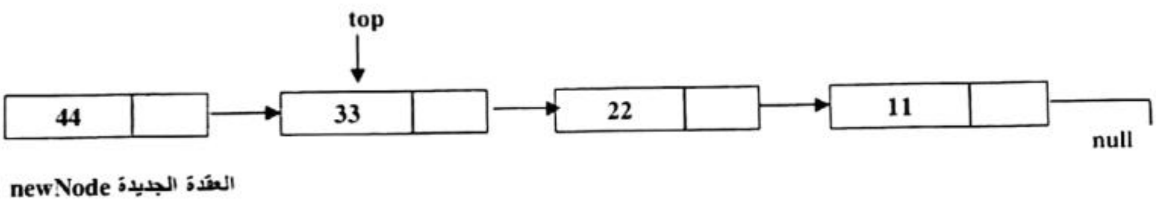
```
public void push (Object item)
{
Node newNode = new Node(item, top)
top = newNode;
}
```

والمثال التالي يوضح هذه العملية:

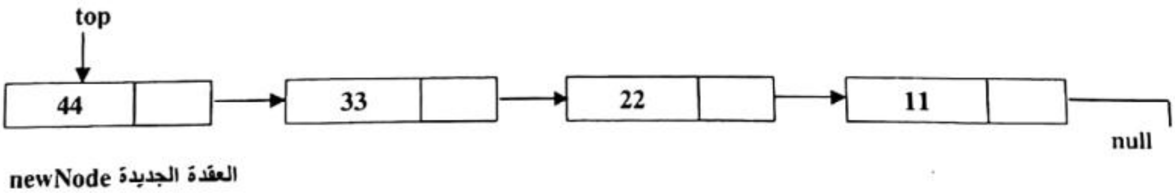
لنفترض أن لدينا المكدس التالي ونريد إضافة عقدة جديدة عند قمته.



1- ننشى عقدة جديدة newNode وندخل بيانات العقدة الجديدة (لتكن مثلا: 44)، وكذلك المؤشر (حيث يشير مؤشر العقدة الجديدة إلى ما يؤشر إليه القمة(top)).



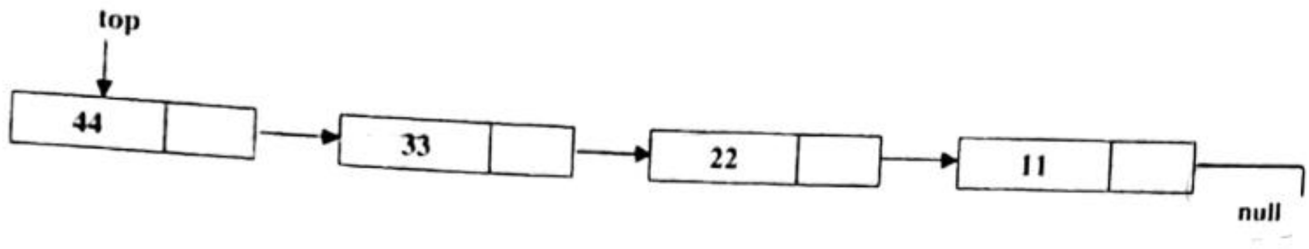
2- نحدث مؤشر عقدة القمة top ليشير إلى العقدة الجديدة (newNode).



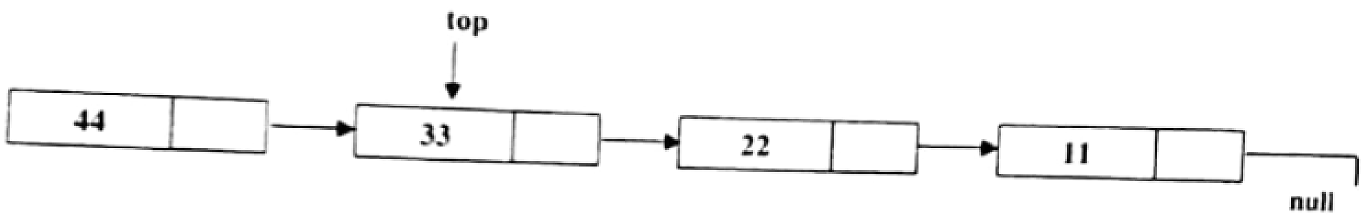
حذف عقدة من المكدس pop

عملية حذف عقدة من المكدس تتم أيضا عند قمة المكدس، أي أخر عنصر تم إدخاله سيتم حذفه أولا. تتم عملية الحذف بجعل مؤشر القمة يشير إلى العقدة التالية (أي يؤشر إلى ما يؤشر إليه جزء الـnext للعقدة الواقعة في قمة المكدس). لاحظ أن منهج الـpop سيقوم بإرجاع جزء البيانات data من العقدة الواقعة في قمة المكدس قبل حذفها. أما بالنسبة للعقدة القديمة والتي كانت في قمة المكدس، فإن المنظف في الجافا garbage collector سوف يتخلص منها ويعيدها للذاكرة مرة أخرى، فلا داعي لفعل أي شيء بالنسبة لها. فقط هناك استثناء سيتم إلقاءه في حال محاولة الحذف من المكدس الفارغ. والمثال التالي يوضح هذه العملية:

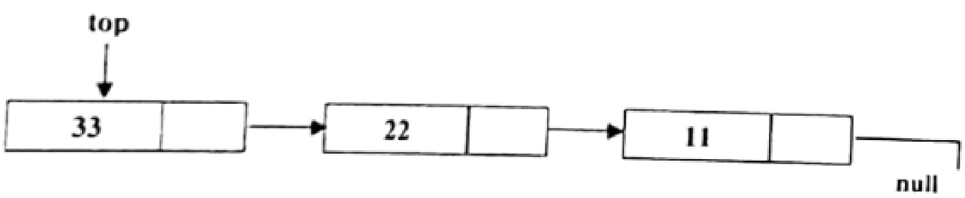
لنفترض أن لدينا المكدس التالي ونريد القيام بعملية الحذف عليه pop().



1- نغير مؤشر الـtop ليشير إلى العقدة التالية (أي إلى ما يشير إليه جزء الـnext للعقدة الواقعة في قمة المكدس (بمعنى ***top.next***))، كالتالي:



2- المكدس سيصبح كالتالي بعد أن يقوم الـ garbage collector في الجافا بحذف العقدة القديمة:



والمنهج التالي يقوم بهذه العملية:

```
public Object pop() throws Exception
{
    Object element = null;
    if(isEmpty())
        throw new Exception("(1): The stack is Empty.");
    else
    {
        element = top.date;
        top = top.next;
    }
    return element;
}
```

أما للحصول على البيانات data الواقعة بقمة المكدس دون حذفها يمكننا استخدام المنهج ()getTop والذي يرجع مؤشر لقمة المكدس ومن ثم الوصول لجزء البيانات الخاص بها.

منهج هل المكدس فارغ isEmpty

يكون المكدس فارغ عندما يكون مؤشر الtop مساوياً لـnull، والمنهج يكون كالتالي:

```
public boolean isEmpty()
{
     return (top == null);
}
```

منهج زيارة "طباعة" عناصر المكدس printStack

يمكننا المرور أو زيارة كل عقدة من عقد المكدس ابتداءً من عقدة القمة وحتى الوصول إلى أخر عقدة وذلك لإجراء أية عملية معالجة أخرى أو لطباعتها وكالتالي:

```
public void printStack( )
{
     for (Node p = top ; p != null; p = p.next)
     {
          System.out.println("Data : " + p.data);
     }
}
```

التالي هو البرنامج كاملا محتويا على كل المناهج سالفة الذكر.

```java
public class StackClass
{
    protected Node top;

    public StackClass()
    {
        top = null ;
    }
    public void push (Object item)
    {
        Node newNode = new Node (item, top);
        top = newNode;
    }
    public Object pop() throws Exception
    {
        Object element = null;

        if(isEmpty())
            throw new Exception("(1): The stack is Empty.");
        else
        {
            element = top.data;
            top = top.next;
        }
        return element;
    }
    public boolean isEmpty()
    {
        return (top == null);
    }
    public Node getTop()
    {
        return top;
    }
    public void printStack()
    {
        for (Node p = top ; p != null; p = p.next)
        {
            System.out.println("Data : " + p.data);
        }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل:

```
public class UseStack
{
    public static void main(String[] args)
    {
        System.out.println("\f");

        try
        {
            StackClass stack = new StackClass();
            stack.push(10);
            stack.push(20);
            stack.push(30);
            stack.push(40);
            stack.push(50);

             // System.out.println(stack.pop() + " at the top
        is removed.");
             // System.out.println(stack.pop() + " at the top
        is removed.");
             // System.out.println(stack.pop() + " at the top
        is removed.");
             // System.out.println(stack.pop() + " at the top
        is removed.");
             // System.out.println(stack.pop() + " at the top
        is removed.");
             // System.out.println(stack.pop() + " at the top
        is removed.");
        }
        catch (Exception e)
        {
            System.out.println("ERROR" + e.getMessage());
        }
    }
}
```

تطبيقات المكدس

يستخدم المكدس في العديد من التطبيقات، منها على سبيل المثال:

1- تمثيل التعابير الحسابية وإيجاد قيمها.
2- موازنة أقواس التعابير الحسابية.
3- استدعاء البرامج الفرعية.
4- الإستدعاء الذاتي للمناهج.

أولا تمثيل التعابير الحسابية

التعبير الحسابي المتمثل في مجموعة من المتغيرات و/أو الثوابت (***opersnds***) يتوسطه مجموعة من العمليات الحسابية (***operators***) والمتمثلة في: + ,- ,x ,/، هي واحدة من عدة صيغ يمكن أن يمثل بها التعبير الحسابي، ويطلق على هذه الصيغة بالتعبير الوسطي (***infix***). كمثال: a+b، أي أن العملية الحسابية (+) تتوسط المتغيرات a وb.

في الواقع إن استخدام التعبير الوسطي قد يتطلب وقت ومعالجة أكبر من المترجمات، الأمر الذي أدى لتحويل التعبير الوسطى إلى إحدى التعبيرين التاليين:

1- الصيغة القبلية (***prefix***).
2- الصيغة البعدية (***postfix***).

ففي الصيغة القبلية تسبق الإشارة المتغيرات، فالصيغة القبلية للمثال السابق هي: ab+. أما في الصيغة البعدية فالإشارة تتبع المتغيرات، أي يصبح التعبير الحسابي السابق بالصيغة البعدية كما يلي: ab+.

ولكن كيف يتم تحويل الصيغة الوسطية إلى صيغة قبلية أو بعدية في التعابير الحسابية ومن ثم وقراءتها؟

تحويل الصيغة الوسطية (*infix*) إلى الصيغة القبلية (*prefix*)

لنفترض أن لدينا التعبير الحسابي التالي في الصيغة الوسطية ***infix***:

```
A + B * C
```

لتحويله إلى صيغة *prefix* نتبع الخطوات التالية:

❖ وضع الأقواس حسب الأولويات

```
( A + ( B * C ) )
```

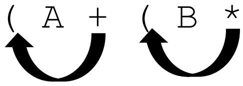
(عدد الأقواس اليمنى = عدد الأقواس اليسرى = عدد العمليات الحسابية)

❖ حذف الأقواس اليمنى

```
( A + ( B * C
```

❖ استبدال الأقواس اليسرى بالعمليات المرتبطة بها.

```
( A + ( B * C
```



فيكون التعبير الحسابي في الصيغة القبلية كالتالي:

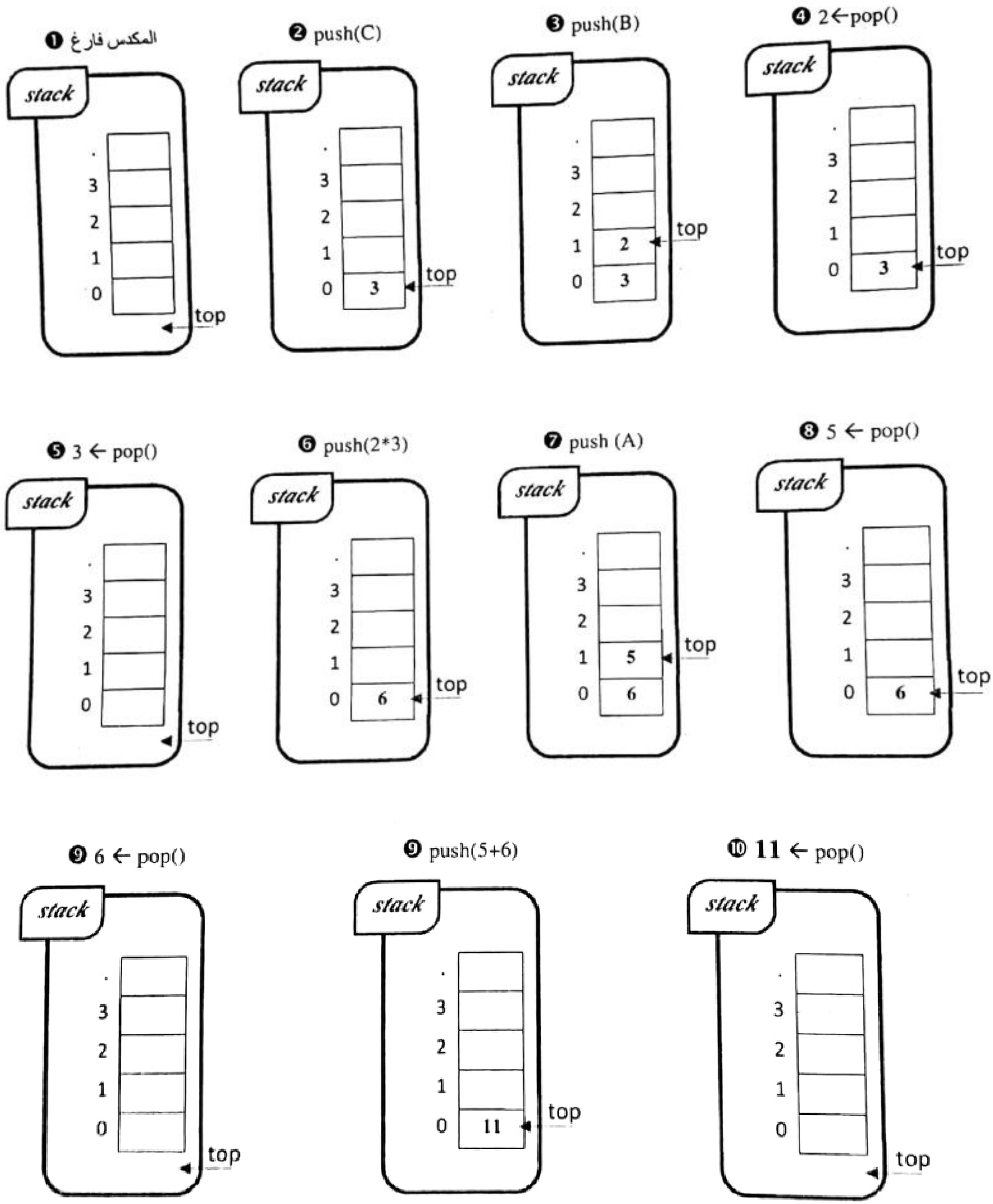
```
+A*BC
```

الآن سنقوم باستخدام المكدس لقراءة وإيجاد نتيجة هذا التعبير (أي في صيغة الـ***Prefix***):

أولا نقوم بقراءة التعبير من اليمين إلى اليسار وعند مرورنا على معامل (أي متغير أو ثابت) نقوم بإجراء عملية الدفع (***push***) إلى المكدس، أما عند مرورنا على علامة حسابية فنقوم بإجراء عملية السحب (***pop***) لمرتين متتاليتين من المكدس وإجراء هذه العملية الحسابية على القيم المرجعة من عمليتي الـ***pop***، ثم نقوم بعملية دفع النتيجة إلى المكدس مرة أخرى. عند الإنتهاء من كل العمليات سيكون هناك قيمة وحيدة بالمكدس هي ناتج التعبير الحسابي.

لنفترض أن:

```
A = 5
B = 2
C = 3
```



نتيجة التعبير الحسابي هي (11)

مثال آخر: حول التعبير الحسابي التالي من صيغة الـ***infix*** إلى صيغة الـ***prefix*** وأوجد نتيجة التعبير بعد التحويل مستخدماً المكدس في ذلك (افترض أن: X=2, Y=3, Z=4, W=7).

(W+5)/((X+Y-Z)*6)

خطوات التحويل:

❖ وضع الأقواس حسب الأولويه:

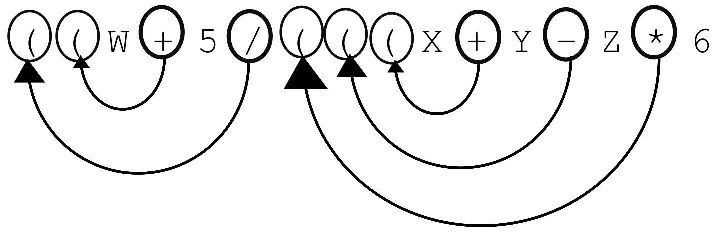
((W + 5) / (((X + Y) - Z) * 6))

(عدد الأقواس اليمنى = عدد الأقواس اليسرى = عدد العمليات الحسابية)

❖ حذف الأقواس اليمنى

((W + 5 / (((X + Y - Z * 6

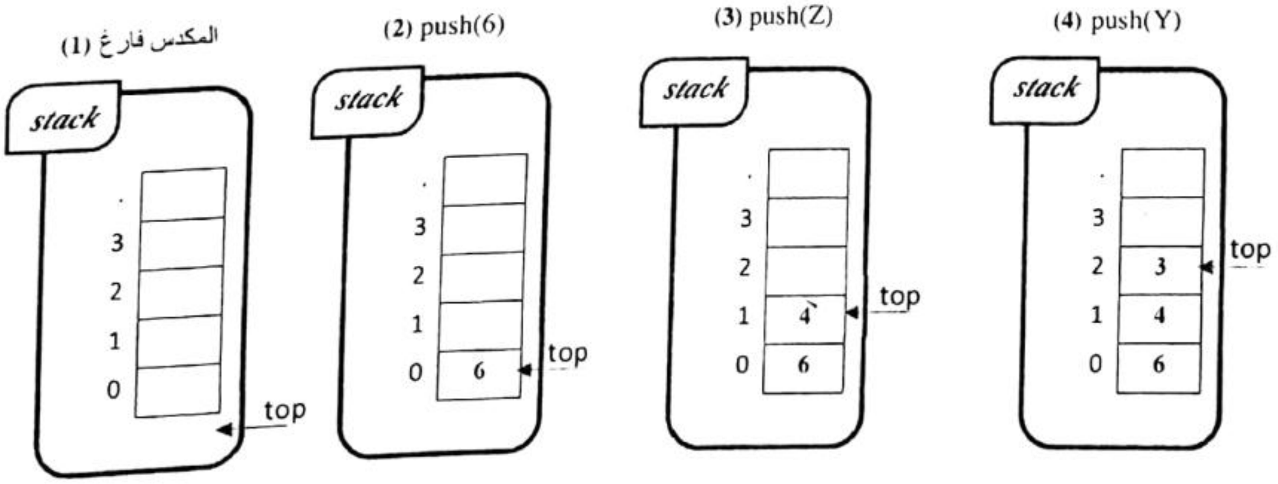
❖ استبدال الأقواس اليسرى بالعمليات المرتبطة بها.



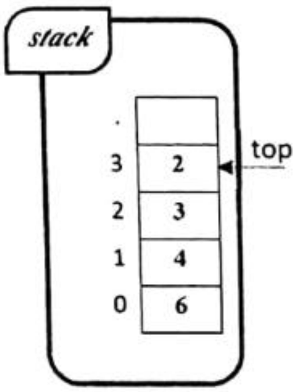
فيكون التعبير الحسابي في الصيغة القبلية كالتالي:

/ + W 5 * - + X Y Z 6

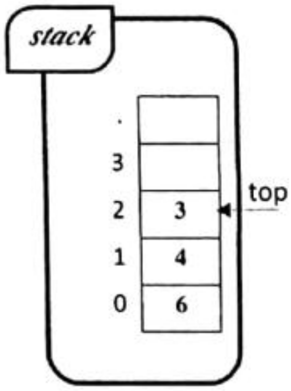
الان سنقوم باستخدام المكدس لقراءة وإيجاد نتيجة هذ التعبير (أي في صيغة الـ***Prefix***) وذلك بقراءة التعبير من اليمين إلى اليسار وإجراء عمليات الدفع والسحب كما سبق شرحه وعلى النحو التالي:



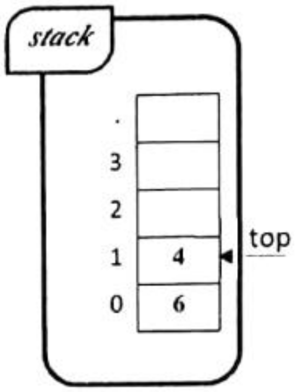
(5) push(X)



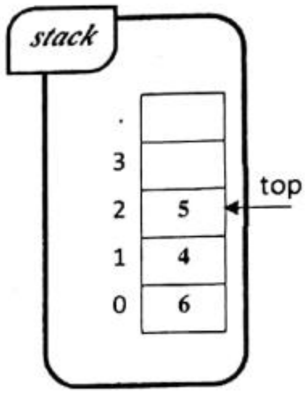
(6) 2 ← pop()



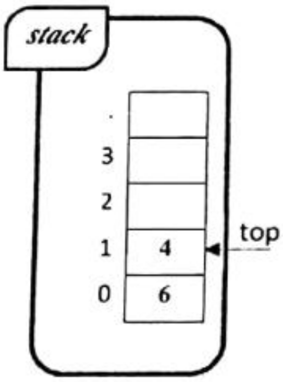
(7) 3 ← pop ()



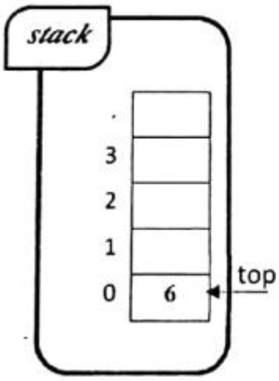
(8) push(2+3)



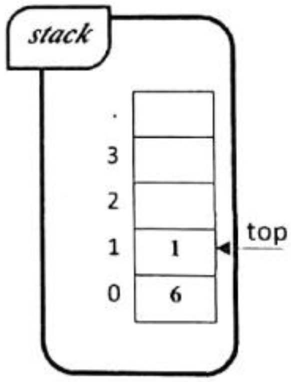
(9) 5 ← pop()



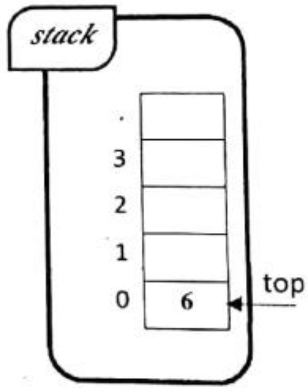
(10) 4 ← pop()



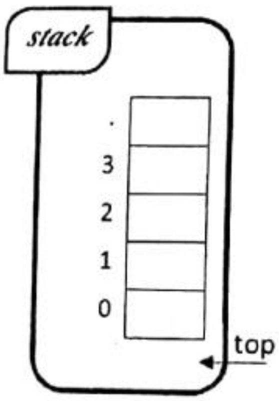
(11) push (5-4)



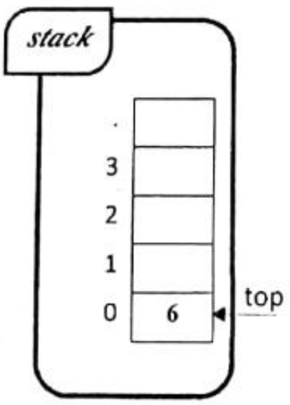
(12) 1 ← pop()



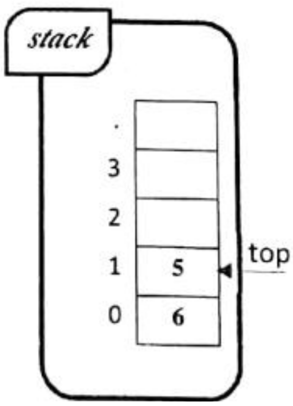
(13) 6 ← pop()



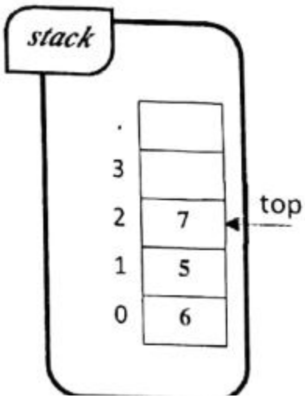
(14) push(1*6)



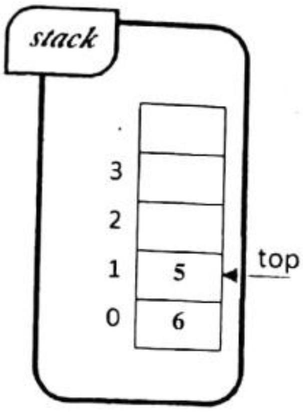
(15) push (5)



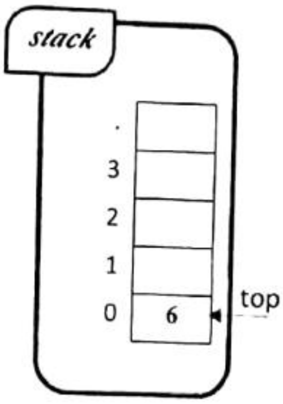
(16) push(W)



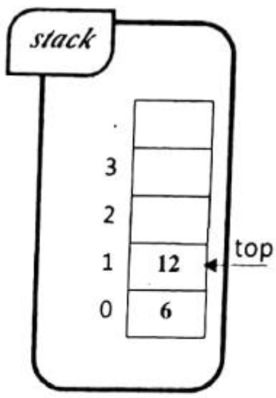
(17) 7 ← pop()



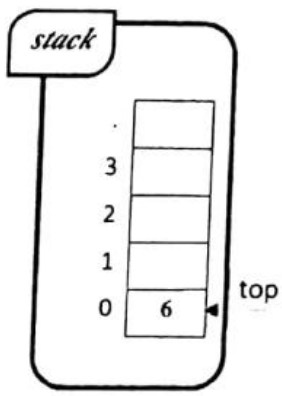
(18) 5 ← pop()

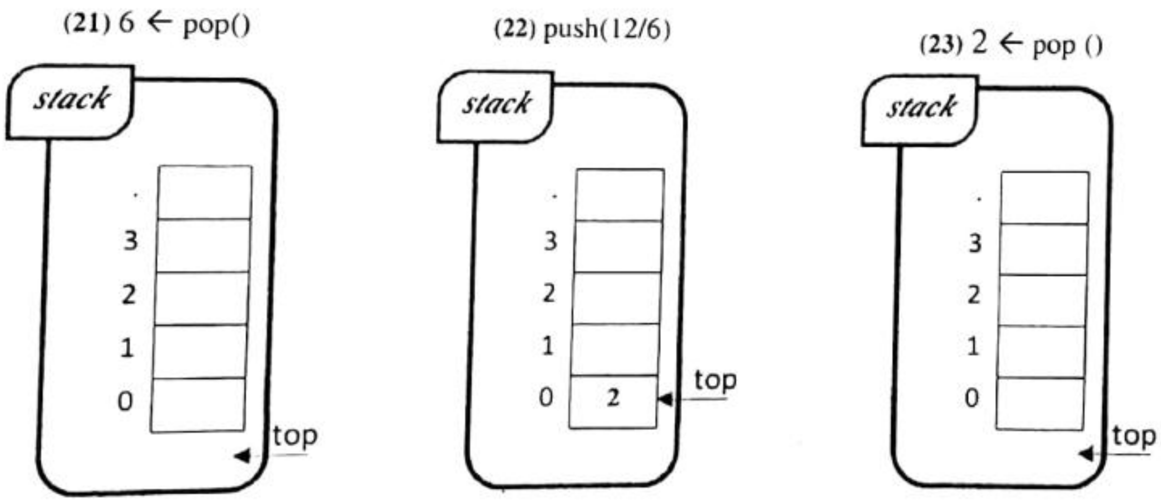


(19) push (7+5)



(20) 12 ← pop()





نتيجة التعبير الحسابي هي (2)

تحويل الصيغة الوسطية (*infix*) إلى الصيغة البعدية (*postfix*)

لنفترض أن لدينا التعبير الحسابي التالي في الصيغة الوسطية ***infix***:

A + B * C

لتحويله إلى صيغة ***postfix*** نتبع الخطوات التالية:

- ❖ نضع الأقواس حسب الأولويه

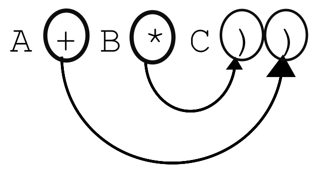
(A + (B * C))

(عدد الأقواس اليمنى = عدد الأقواس اليسرى = عدد العمليات الحسابية)

- ❖ نحذف الأقواس اليسرى

A + B * C))

- ❖ نستبدل الأقواس اليمنى بالعمليات المرتبطة بها.



فيكون التعبير الحسابي في الصيغة البعدية كالتالي:

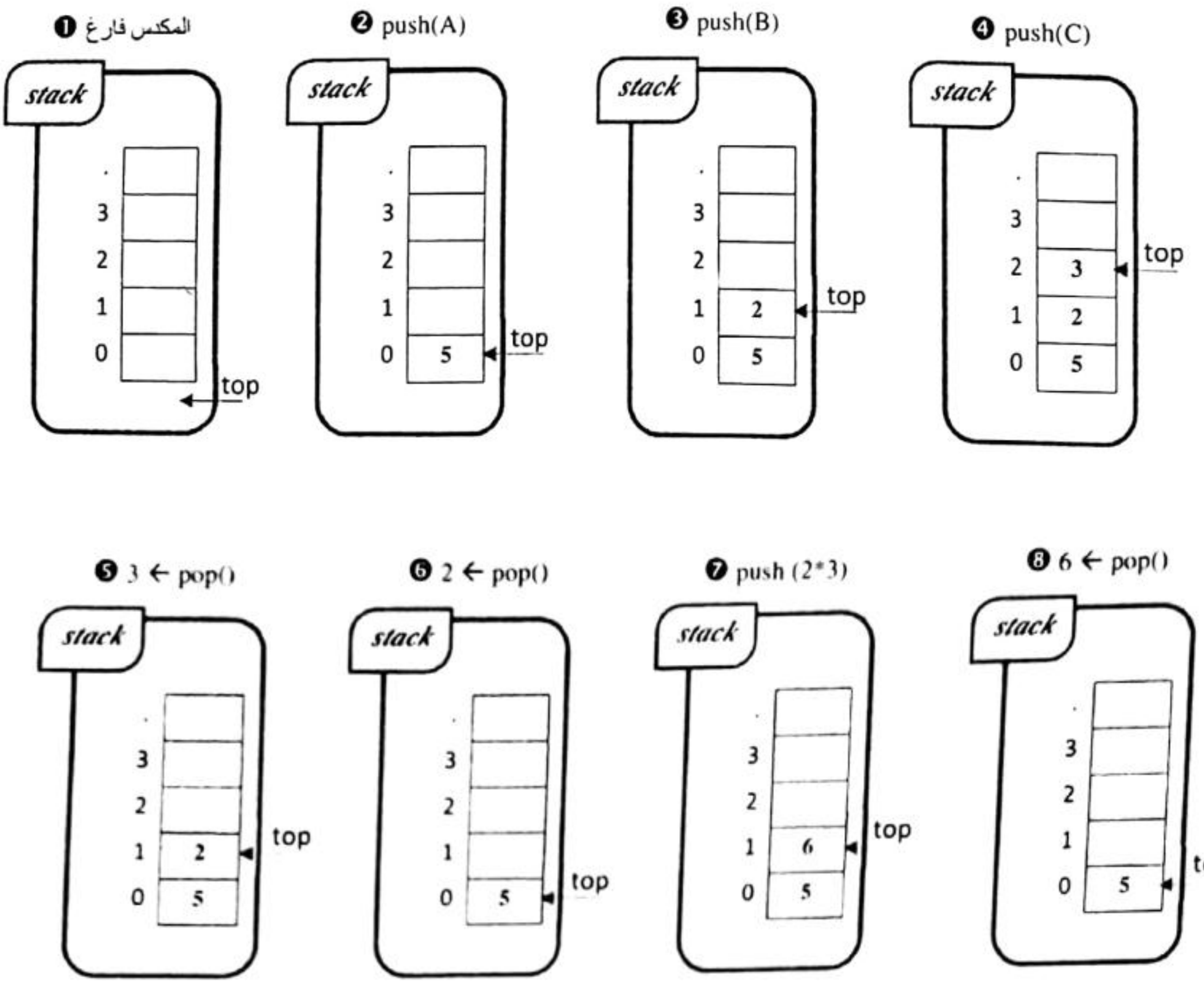
ABC*+

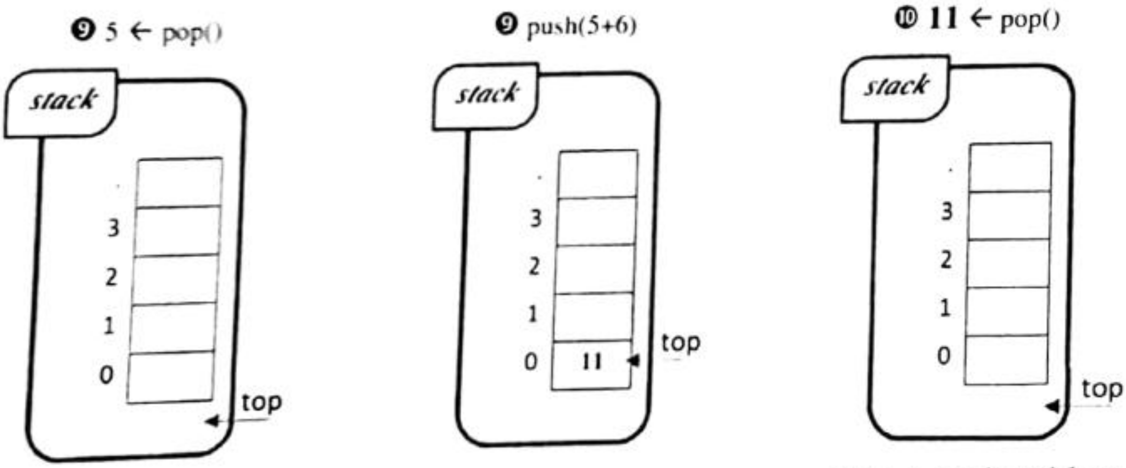
الآن سنقوم باستخدام المكدس لقراءة وإيجاد هذا التعبير (أي في صيغة الـ***Postfix***):

أولا نقوم بقراءة التعبير من اليسار إلى اليمين وعند مرورنا على معامل (أي متغير أو ثابت) نقوم بإجراء عملية الدفع (***push***) إلى المكدس، أما عند مرورنا على علامة حسابية فنقوم بإجراء عملية السحب (***pop***) لمرتين متتاليتين من المكدس وإجراء هذه العملية الحسابية على القيم المرجعة من عمليتي الـ***pop***، ثم نقوم بعملية دفع النتيجة إلى المكدس مرة أخرى. عند الإنتهاء من كل العمليات سيكون هناك قيمة واحدة بالمكدس، هي ناتج التعبير الحسابي.

لنفترض أن:

```
A = 5
B = 2
C = 3
```





نتيجة التعبير الحسابي هي (11)

مثال أخر:

حول التعبير الحسابي التالي من صيغة الـinfix إلى صيغة الـpostfix.

```
(W+5)/((X+Y-Z)*6)
```

خطوات التحويل:

❖ وضع الأقواس حسب الأولويات

```
( ( W + 5 ) / ( ( ( X + Y ) - Z ) * 6 ) )
```

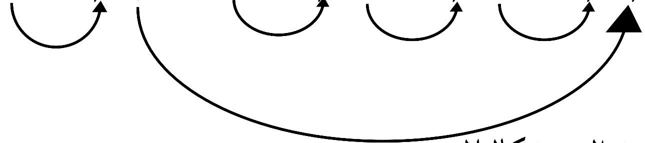
(عدد الأقواس اليمنى = عدد الأقواس اليسرى = عدد العمليات الحسابية)

❖ حذف الأقواس اليسرى

```
W + 5 ) / X + Y ) - Z ) * 6 ) )
```

❖ استبدال الأقواس اليمنى بالعمليات المرتبطة بها.

```
W + 5 ) / X + Y ) - Z ) * 6 ) )
```



فيكون التعبير الحسابي في الصيغة البعدية كالتالي:

```
W5+XY+Z-6*/
```

سنترك لك عملية التأكد من أن نتيجة التعبير بعد التحويل إلى صيغة ***postfix*** هي 2 مستخدما في ذلك المكدس كما سبق شرحه بالأمثلة السابقة (افترض أن: X=2, Y=3, Z=4, W=7).

ثانيا موازنة أقواس التعابير الحسابية

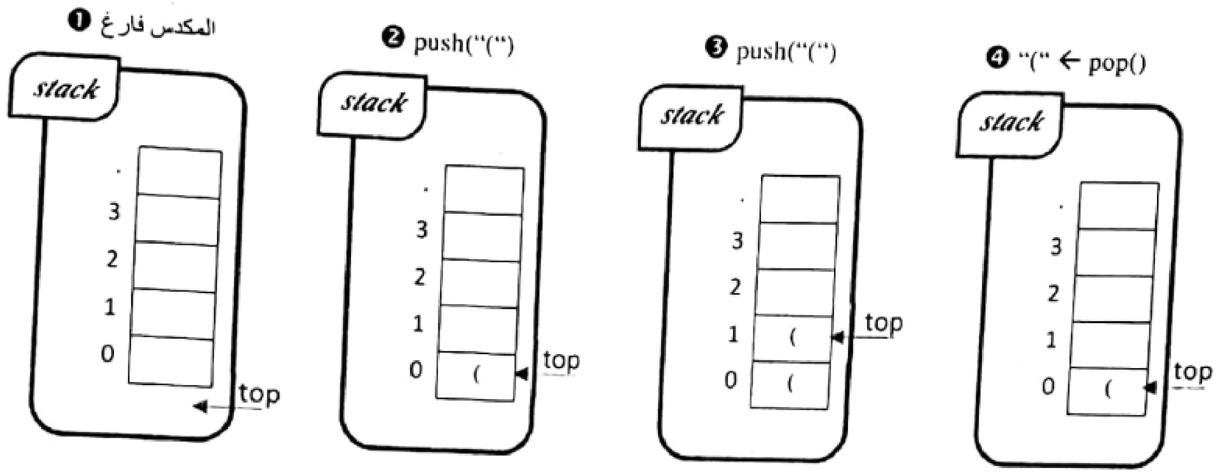
عند كتابة البرامج ضمن إحدى لغات البرمجة سيقوم المترجم بالتأكد من أن أقواس التعابير الحسابية متوازنة، أي أن كل قوس أيسر يقابله قوس أيمن، وبالإجمالي فإن عدد الأقواس اليسرى تساوي عدد الأقواس اليمنى.

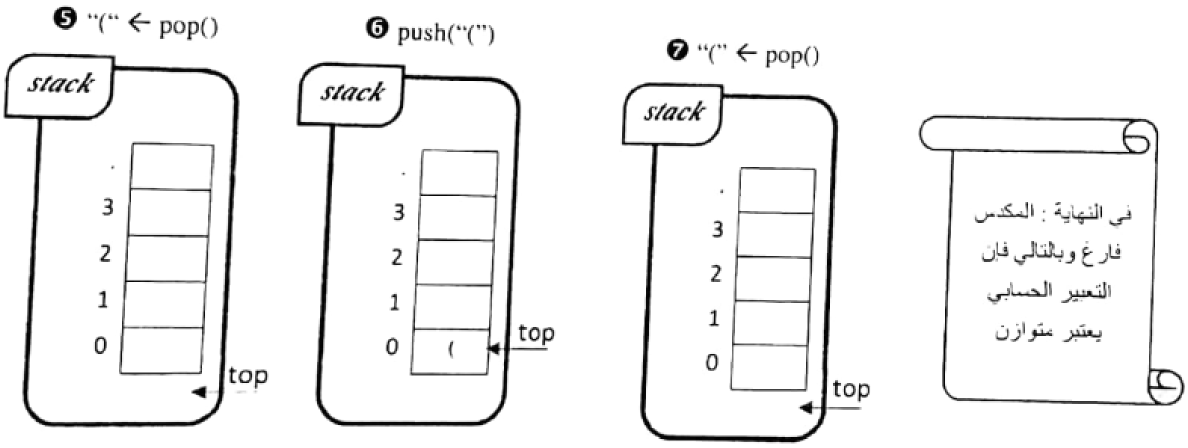
فمثلا:

```
((x+y)*2)-(z/6)  //  يعتبر تعبير حسابي متوازن
(x+y)-(z/6       //  يعتبر تعبير حسابي غير متوازن
(x+y)-z/6)       //  يعتبر تعبير حسابي غير متوازن
```

يستخدم المكدس للتأكد من أن عدد الأقواس ضمن التعابير الحسابية متوازنة ويتم ذلك بالمرور على التعبير الحسابي من اليسار لليمين: في حالة المرور على قوس أيسر يتم دفعه (***push()***) إلى المكدس، وفي حال المرور على قوس أيمن يتم إجراء عملية سحب (***pop()***) – النتيجة وبعد إجراء كل عمليات السحب: في حال أيضا التعبير يكون غير متوازنا في حال محاولة إجراء عملية سحب ***pop()*** على مكدس فارغ. سنقوم الآن بتطبيق هذا الإجراء على الأمثلة السابقة، ففي حالة التعبير الحسابي التالي:

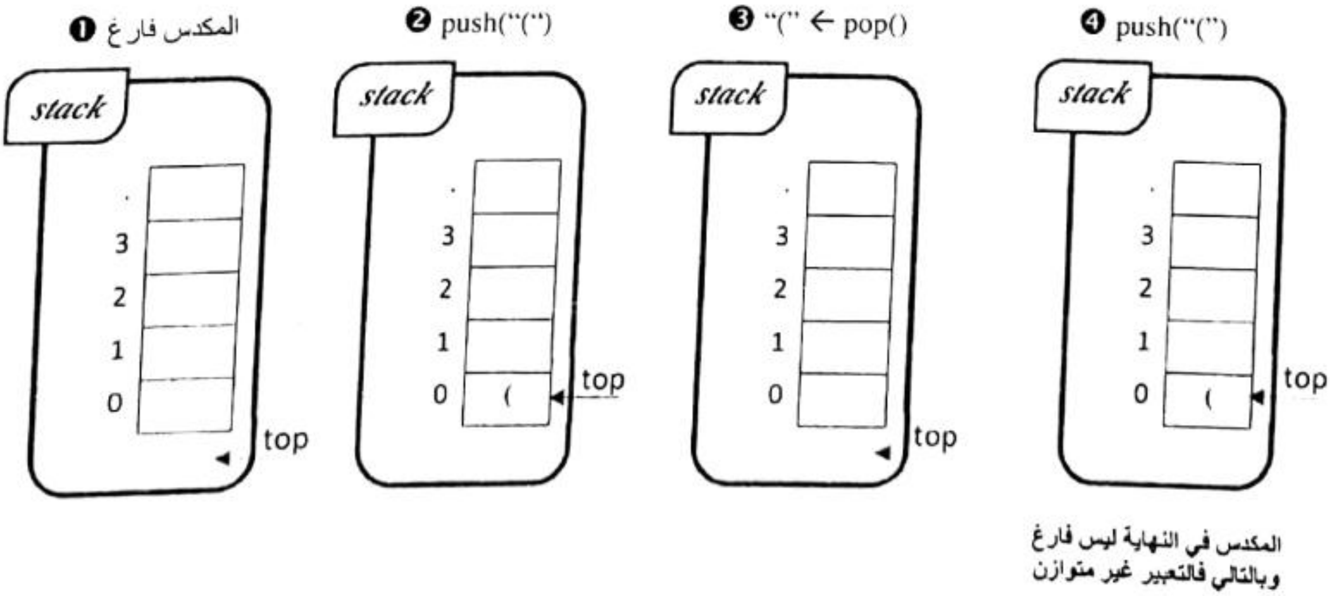
```
((x+y) *2) - (z / 6)
```





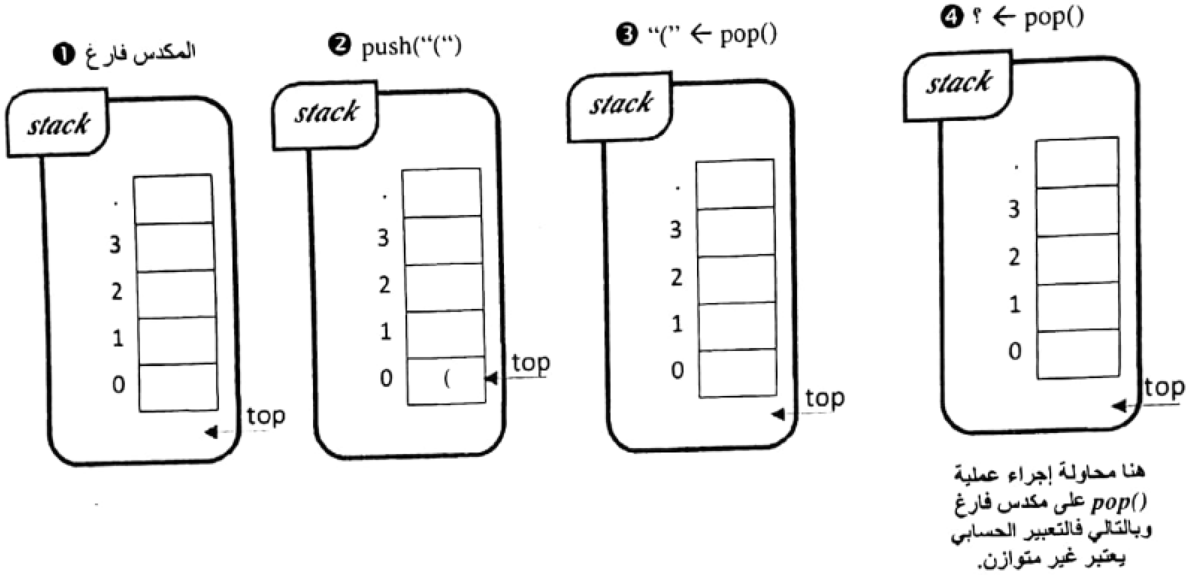
أما في حالة التعبير الحسابي:

(x+y) - (z / 6



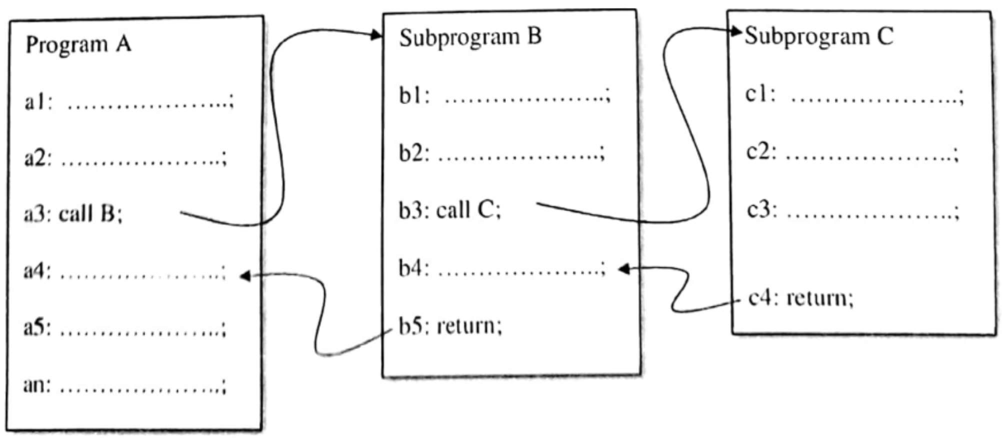
أما في حالة التعبير الحسابي التالي:

```
(x+y) - z / 6)
```

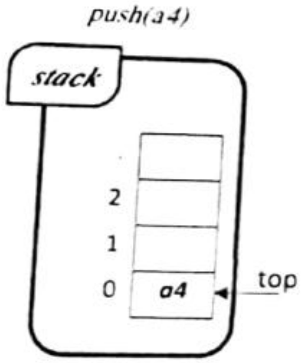


ثالثا استدعاء البرامج الفرعية

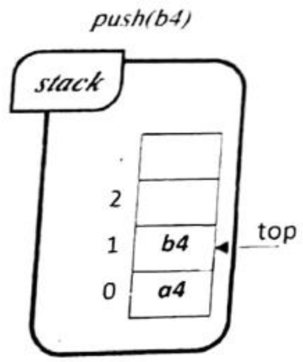
عادة ما يتم تقسيم البرامج الكبيرة إلى برامج فرعية صغيرة، يؤدي كل برنامج فرعي وظيفة معينة، ويسمى البرنامج الفرعي بالمنهج "method" أو الدالة "function" أو الإجراء أو الروتين. يستخدم المكدس عادة في بناء المترجمات للحفاظ على سير البرنامج والتنقل بين البرنامج الرئيسي والبرامج الفرعية. فلتوضيح الفكرة لنفترض أن لدينا البرنامج التالي والمكون من البرنامج الرئيسي A وبرنامجين فرعيين B و C. هنا سوف نوضح عمل المكدس في الإنتقال والعودة من وإلى البرامج الفرعية بصورة مبسطة ولن نركز على التفاصيل الأخرى.



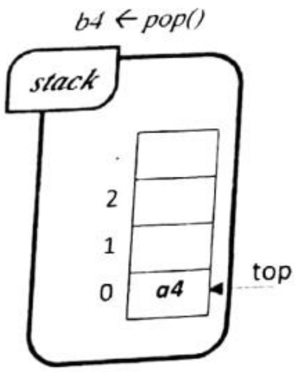
منطق تنفيذ البرنامج السابق هو أن البرنامج الرئيسي يقوم بتنفيذ الجمل بشكل تتابعي من أول جملة إلى أن يصل إلى نقطة استدعاء البرنامج الفرعي B عند الجملة "a3" حيث ينتقل إلى تنفيذ الجمل هناك بشكل تتابعي أيضا إلى أن يصل إلى جملة الإستدعاء الثانية عند الجملة "b3" حيث يستدعي البرنامج الفرعي C فينتقل التنفيذ إلى البرنامج الفرعي C من بدايته وينفذ الجمل إلى أن يصل إلى جملة الرجوع return ليرجع التنفيذ بعد ذلك إلى الجملة "b4" في البرنامج الفرعي B ويكمل تنفيذ بقية الجمل هناك إلى أن يصل إلى جملة الرجوع return في البرنامج الفرعي B فيرجع التنفيذ بعد ذلك إلى البرنامج الرئيسي A عند الجملة "a4" ليكمل تنفيذ بقية البرنامج الرئيسي.



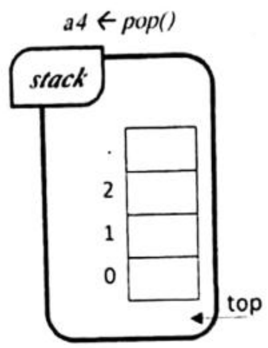
الفكرة من استخدام المكدس هنا هو الإحتفاظ بعناوين الجمل التالية لجمل الإستدعاء والتي يجب الرجوع إليها عن الإنتهاء من تنفيذ البرامج الفرعية في السيناريو السابق، وعند وصول البرنامج الرئيسي لجملة استدعاء البرنامج الفرعي ***B*** عن الجملة "***a3***" فيقوم بعملية دفع للجملة التالية لها وهي "***a4***" في المكدس ليصبح المكدس كما بالشكل التالي:



ثم ينتقل التنفيذ إلى البرنامج الفرعي ***B*** ويتم تنفيذ الجمل إلى أن يصل لجملة استدعاء البرنامج الفرعي ***C*** عند الجملة "***b3***" هنا يتم الاحتفاظ بعنوان الجملة التالية لها وهي "***b4***" ودفعها في المكدس وهي الجملة التي يجب الرجوع إليها عند إكمال تنفيذ البرنامج الفرعي ***C*** كما بالشكل التالي:



ثم عند الإنتهاء من تنفيذ البرنامج الفرعى ***C*** والوصول إلى جملة الـ***return*** يتم إجراء عملية سحب ***pop( )*** للمكدس وانتقال التنفيذ إلى عنوان الجملة التي كانت في قمة المكدس وهي هنا "***b4***". فيقوم البرنامج بإكمال تنفيذ البرنامج الفرعى ***B***، فيصبح المكدس كما بالشكل التالي:



وعند وصول التنفيذ لجملة الرجوع ***return*** في البرنامج الفرعي ***B*** يتم إجراء عملية سحب ***pop()*** من المكدس وينتقل التنفيذ إلى عنوان الجملة التي كانت في قمة المكدس وهي هنا: "***a4***" ليتم إكمال تنفيذ بقية الجمل في البرنامج الرئيسي ابتداءً من هذه الجملة إلى نهاية البرنامج، فيصبح المكدس كما بالشكل.

رابعاً الإستدعاء الذاتي للمناهج Recursion

هو استدعاء البرنامج الفرعي لنفسه عدة مرات حتى يتحقق شرط معين. فالمكدس يستخدم هنا ليحتفظ بالقيم واسترجاعها عند توقف البرنامج الفرعي من استدعاء نفسه.

سنقوم بتوضيح الاستدعاء الذاتي من خلال المثال التالي: لنفترض أننا نريد إيجاد مجموع الأعداد الصحيحة من 1 إلى 5. فبدون استخدام الاستدعاء الذاتي يمكننا كتابة المنهج بشكله البسيط كالتالي:

```
public int sum(int n)
{
     int s = 0;
     for (int i = 1; i <= n; i++)
          s = s + I;
     return s;
}
```

لاستدعاء هذا المنهج نكتب:

```
int sum1_5 = sum (5);
```

لكن يمكننا كتابة برنامج لإيجاد المجموع باستخدام استدعاء المنهج لنفسه كالتالي:

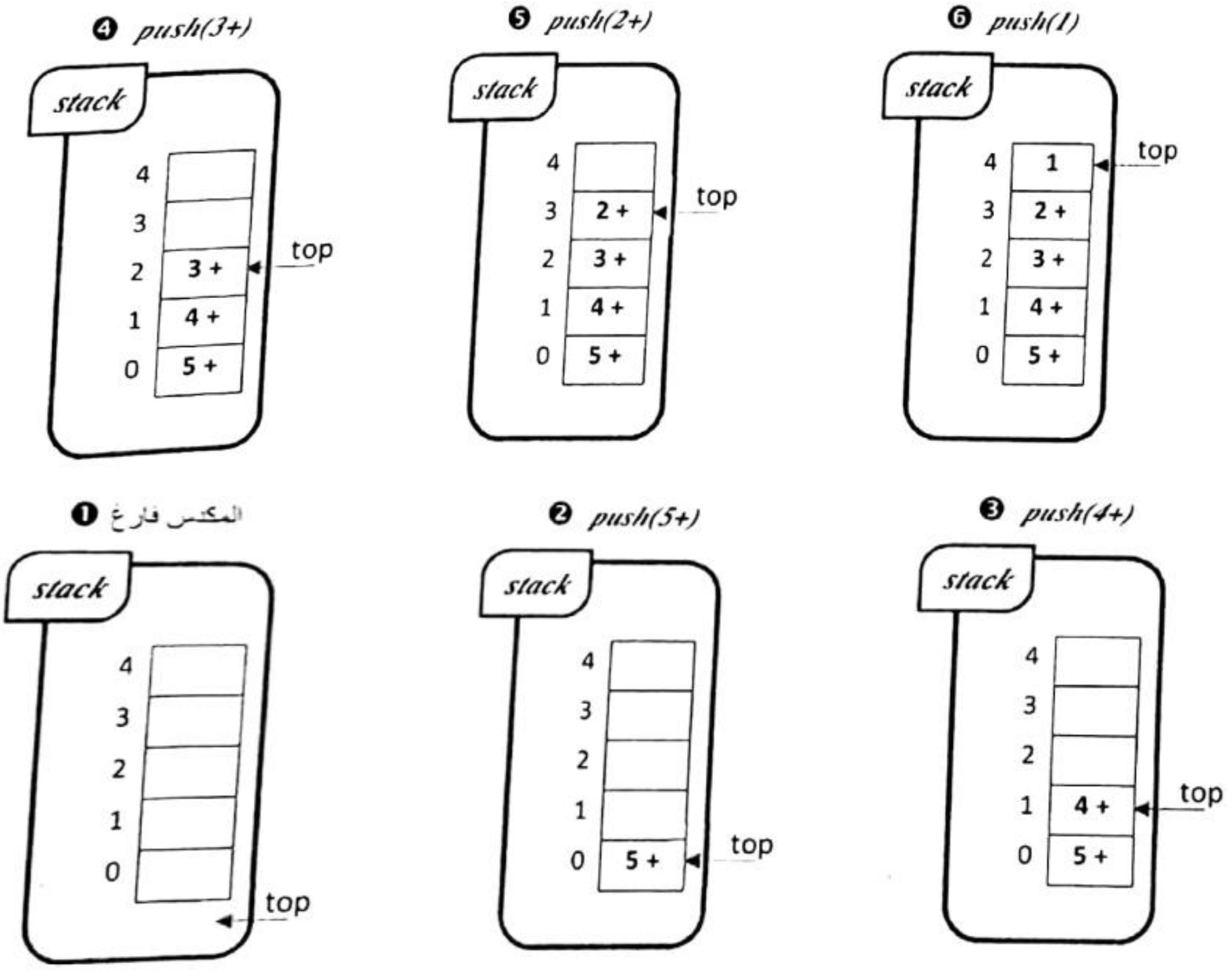
```
public int sum(int n )
{
     if ( n == 1 )
          return 1;
     else
          return (n + sum(n-1));
}
```

لاستدعاء هذا المنهج نكتب نفس الأمر السابق.

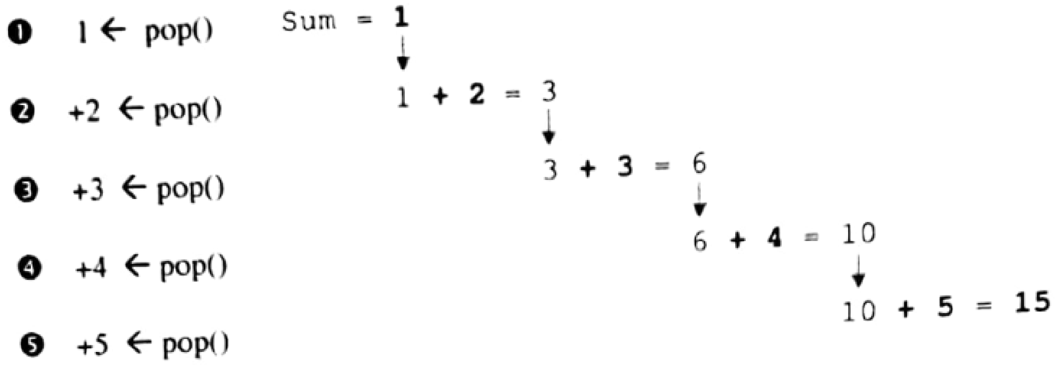
هذا المنهج يستدعي نفسه مع إنقاص 1 من قيمة ***n*** في كل مرة إلى أن يتحقق الشرط. أي بمعنى أدق في المرة الأولى يقوم المنهج باستدعاء نفسه لإيجاد مجموع الأرقام من 1 إلى 4 مع الإحتفاظ بقيمة 5 في المكدس (أي إجراء عملية ***push()***)، وفي المرة الثانية يستدعي المنهج نفسه لإيجاد مجموع الأرقام من 1 إلى 3 مع الإحتفاظ بالرقم 4 في المكدس، وهكذا حتى يتم الإنتهاء من استدعاء المنهج لنفسه بتحقق الشرط وهو أن يكون قيمة ***n*** مساوية للواحد. لابد أن يكون هناك شرط لإيقاف استدعاء المنهج لنفسه وهو هنا: (n == 1) وإلا فإن المنهج سيستدعي نفسه إلى ما لا نهاية، يمكننا تمثيل عملية إيجاد مجموع الأعداد من 1 إلى 5 كالتالي:

$$\begin{aligned} \sum 5 &= 5 + \sum 4 \\ &= 5 + 4 + \sum 3 \\ &= 5 + 4 + 3 + \sum 2 \\ &= 5 + 4 + 3 + 2 + 1 \end{aligned}$$

وأن شكل المكدس سيكون على النحو التالي:



تتم عملية إيجاد المجموع بعمليات سحب ***pop()*** لما تم الإحتفاظ به في المكدس وعلى النحو التالي:



مثال أخر: اكتب منهج اسمه `factorial()` يقوم بإيجاد مضروب عدد ما معطى كمعلمة مستخدماً في ذلك الاستدعاء الذاتي للمنهج. علما بأن مضروب العدد هو:

```
n! =(n-1)x(n-2)x(n-3)x ... x 3 x 2 x 1
```

فمثلا مضروب العدد !5 هو:

```
5!=5x4x3x2x1
```

أي

```
5! = 5x4!
   = 5x4x3!
   = 5x4x3x2!
   = 5x4x3x2x1!
```

```
public long factorial(int n)
{
if (n == 1)
     return 1;
else
     return (n * factorial(n-1));
}
```

يمكن استدعاء هذا المنهج لإيجاد مضروب العدد 5 على سبيل المثال كالتالي:

```
long fact_5 = factorial(5);
```

مثال آخر: مستخدما الاستدعاء الذاتي للمنهج، اكتب منهجا لإيجاد مجموع المتسلسلة التالية:

$$S = 1 + \frac{1}{3} + \frac{1}{9} + \frac{1}{27} + \cdots + \frac{1}{3^n}$$

حيث ***n*** يتم استقبالها كمعلمة في رأس المنهج.

أولا لحل هذا السؤال علينا أن نحدد شرط التوقف وهو هنا: عندما ***n*** تساوي صفر فإن المنهج سيرجع 1. لذلك يمكننا كتابة المنهج على النحو التالي:

```
public double sum(int n)
{
    // هنا هو شرط التوقف
    if (n==0)
        return 1;
    else
        return ( (1.0/Math.pow(3, n)) + sum(n-1) );
}
```

يمكن استدعاء هذا المنهج لإيجاد مجموع المتسلسلة حتى الحد الثالث على سبيل المثال كالتالي:

```
double s = sum(3);    // 1.481481 الناتج سيكون
```

الطابور Queue

الطابور: هو هيكل من هياكل البيانات الخطية فهو يمثل مجموعة من العناصر المنظمة في تسلسل. قد تكون هذه العناصر أعداد صحيحة أو حقيقية أوسلاسل حرفية أو كائنات لأصناف معينة. يعتبر الطابور نوع خاص من القائمة حيث أن عملية الإضافة (***Enqueue***) تتم من الخلف (***Rear***)، وعملية الحذف (***Dequeure***) تتم من الأمام، يعمل الطابور بمبدأ الداخل أولا هو الخارج أولا (***First In First Out - FIFO***)، أي أن أول عنصر يتم إضافته سيكون أول عنصر يتم حذفه. عادة ما يتم تمثيل الطابور باستخدام المصفوفات (***arrays***) أو القوائم المترابطة (***Liriked list***).

سنقوم بتمثيل الطابور باستخدام الطريقتين لإجراء عمليات المعالجة المختلفة منها على سبيل المثال:

- ❖ إضافة عنصر إلى الطابور وتعرف بعملية (***Enqueue***).
- ❖ حذف عنصر من الطابور وتعرف بعملية (***Dequeure***).
- ❖ البحث عن عنصر في الطابور واسترجاعه.
- ❖ زيارة عناصر الطابور بشكل تسلسلي (لأجل الطباعة مثلا).

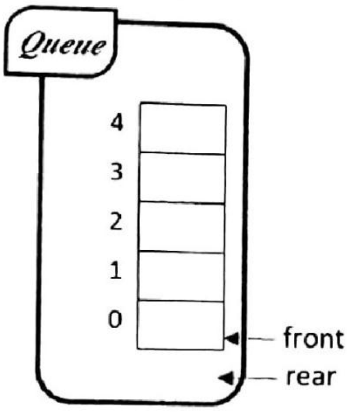
أولا تمثيل الطابور باستخدام المصفوفات

نظرا لأن عملية الإضافة تتم من الخلف وعملية الحذف تتم من الأمام فسيتم استخدام مؤشرين أحدهما اسمه: `front`، والأخر اسمه: `rear`، بالإضافة إلى مصفوفة ذات بعد واحد لتخزين البيانات، هذه المصوفة لها حجم أو سعة إجمالية هي: `capacity`. المصفوفات تستخدم عادة لتمثيل الطابور إما بطريقة خطية أو دائرية، وذلك لأجل بناء إما الطابور الخطي أو الطابور الدائري.

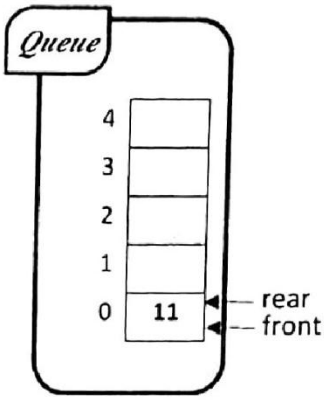
الطابور الخطي

تتم عملية الإضافة في الطابور الخطي بإضافة 1 إلى مؤشر `rear` ووضع البيانات الجديدة في نهاية الطابور إلى أن يتم تعبئة المصفوفة بشكل كامل بالبيانات، أي يصبح `rear` مساويا لـ(`capacity - 1`)، وتتم عملية الحذف بزيادة 1 لمؤشر الـ `front`. شرط الإمتلاء هنا هو (`rear == capacity - 1`)، حتى و إن كان هناك مساحات فارغة في بداية المصفوفة. سنوضح بالأمثلة كيف تتم عمليات الإضافة (Enqueue) والحذف (Dequeue):

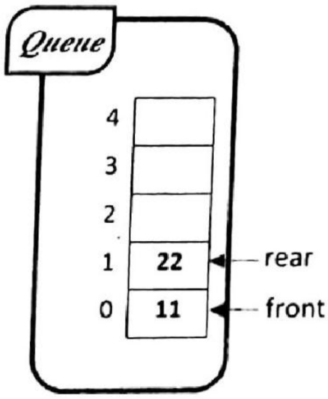
❶ **Empty queue**



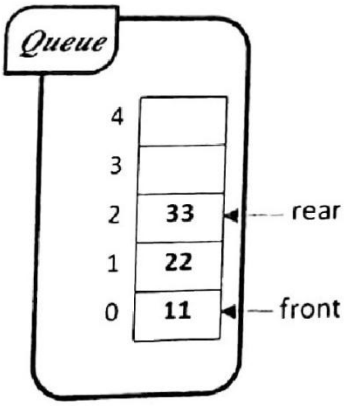
❷ **after enqueue(11)**



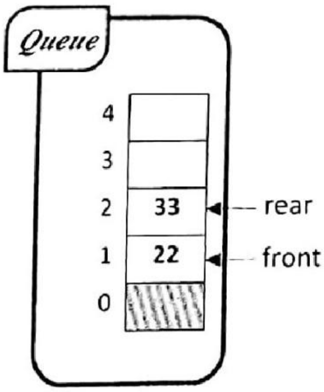
❸ **after enqueue(22)**



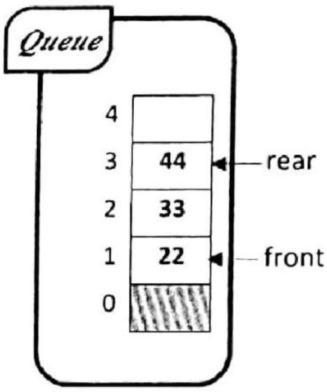
❹ **after enqueue(33)**



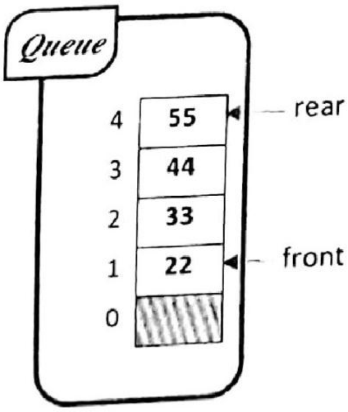
❺ **after dequeue()**



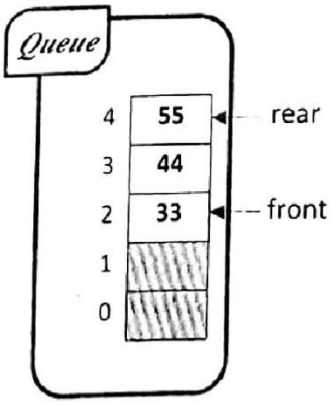
❻ **after enqueue(44)**



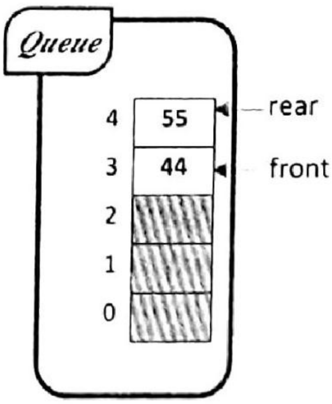
❼ **after enqueue(55)**
The queue is Full



❽ **after dequeue()**
- Full queue



❾ **after dequeue()**
- Full queue



نلاحظ من الأشكال السابقة أن الطابور يعتبر ممتلئ بالرغم من وجود فراغات في بداية المصفوفة بعد عمليات الحذف Dequeue، هذا قد يؤدي إلى إهدار في موارد الذاكرة، ولكن للطابور الخطي أيضا استخداماته، فمثلا: لو أردنا أن نضع قيد على عدد مهام الطباعة التي تستقبلها طابعة مشتركة بين عدة مستخدمين بأن تستقبل فقط 20 مهمة طباعة في اليوم فالطابور الخطي قد يكون هو الأنسب في هذه الحالة؛ حيث يستخدم مصفوفة بها 20 خلية ويضع بها مهام الطباعة ومن ثم يقوم بمعالجة المهام واحدة تلو الأخرى حسب أسبقية قدومها. مثال آخر: إذا كان هناك مصرف يستقبل 50 زبون في اليوم فقط ويغلق أبوابه بعد ذلك، فالطابور الخطي قد يكون مناسباً لهذه الحالة أيضا.

قبل القيام بإجراء العمليات المختلفة على الطابور الخطي باستخدام المصفوفات سنقوم أولا بتكوين فصيل جديد اسمه QueueByArray، يحتوي على تعريف لمصفوفة من الكائنات (هنا سنستخدم الأعداد الصحيحة) اسمها data ذات حجم أوسعة capacity، يتم تحديدها ضمن بانيها، بالإضافة لحقلين واحد اسمه front ويدل على موقع أول عنصر بالطابور، والثاني اسمه: rear ويدل على أخر عنصر بالطابور وكما يلي:

```
public class QueueByArray
{
    private int front;
    private int rear;
    private int capacity;
    private Object[] data;

    public QueueByArray (int c)
    {
        capacity = c ;
        data = new Object[] capacity;
        front = 0 ;
        rear = -1;
    }
}
```

لاحظ أنه عند تكوين كائن جديد من فصيل الطابور فإن مؤشر الـfront سيكون عند 0 ومؤشر الـrear سيكون على 1-، ويتم تحديد سعة الطابور عن طريق القيمة c والتي يتم استقبالها ضمن الباني.

الآن يمكننا استخدام هذا الفصيل ضمن فصيل آخر وليكن: UseQueueByArray، وتكوين كائن من الطابور كالتالي:

```
QueueByArray queue = new QueueByArray(5);
```

هنا حددنا سعة الطابور بـ5 عناصر كحد أقصى. الرقم 5 سيمرر إلى باني الفصيل QueueByArray ليتم تخزينه ضمن الحقل capacity، ثم تحديد طول الطابور بـ5.

منهج إضافة عنصر إلى الطابور الخطي Enqueue

سيقوم بإضافة عنصر جديد يتم استقباله كمعلمة لنهاية الطابور. في حال ما إن كان الطابور ممتلئ فسيتم إلقاء استثناء بذلك. مع كل عملية إضافة جديدة سيتم زيادة 1 لقيمة الحقل rear.

```
public void enQueue (Object object) throws Exception
{
     if (isFull())
          throw new Exception (" (2) : The queue is full !");
     else
     {
          rear = rear +1;
          data[rear] = object;
     }
}
```

يمكننا استدعاء هذا المنهج الآن ضمن الفصيل UseQueueByArray عدة مرات وكالتالي:

```
queue.enQueue(11);
queue.enQueue(22);
```

منهج هل الطابور ممتلئ isFull

لاحظنا من المنهج السابق أننا استخدمنا المنهج ***isFull()*** ليختبر فيما إن كان الطابور ممتلئ أم لا. شرط الإمتلاء يتحقق عندما تكون قيمة الـ rear مساوية لـ 4 أي حجم الطابور الكلي ناقصاً واحد (capacity - 1) فالمنهج كالتالي:

```
public boolean isFull()
{
     return (rear == (capacity - 1));
}
```

منهج هل الطابور فارغ isEmpty

يكون الطابور فارغاً عندما يكون مؤشر الـfront أكبر من مؤشر الـrear، فالمنهج يكون كالتالي:

```
public boolean isEmpty()
{
     return (front > rear);
}
```

منهج حذف عنصر من الطابور Dequeue

تتم عملية حذف العنصر الموجود عند مقدمة الطابور. هذا المنهج سيعيد عنصر البيانات المراد حذفه. لاحظ أنه سيتم زيادة 1 إلى الـfront، وأن استثناء يتم إلقاءه في حال ما إن كان الطابور فارغا.

```
public object deQueue() throws Exception
{
     if (isEmpty())
          throw new Exception (" (1) : The queue is empty !");
     else
     {
          Object obj = data[front];
          data [front] = null;
          front = front + 1;
          return obj;
     }
}
```

يمكنك استدعاء هذا المنهج من الفصيل UseQueueByArray كالتالي:

```
System.out.println( queue.deQueue() + " at the front has been
removed");
```

منهج زيارة "طباعة" عناصر الطابور print

يمكننا المرور أو زيارة كل عنصر من عناصر الطابور وذلك لإجراء أية عملية معالجة أخرى أو لطباعته كالتالي:

```
public void print()
{
     for ( int i = front; i <= rear; i++)
     {
          System.out.println("Data["+1+"] == > " + data[i]);
     }
}
```

يمكن استدعاء منهج الطباعة السابق كالتالي:

```
queue.print();
```

التالي هو البرنامج كاملا ومحتويا على كل المناهج سالفة الذكر.

```
public class QueueByArray
{
    private int front;
    private int rear;
    private int capacity;
    private Object[] data;

    public QueueByArray (int c)
    {
        capacity = c ;
        data = new Object[capacity];
        front = 0 ;
        rear = -1;
    }

    public void enQueue (Object object) throws Exception
    {
        if (isFull())
            throw new Exception (" (2) : The queue is full !");
        else
        {
            rear = rear +1;
            data[rear] = object;
        }
    }

    public Object deQueue() throws Exception
    {
        if (isEmpty())
            throw new Exception (" (1) : The queue is empty
        !");
        else
        {
            Object obj = data[front];
            data [front] = null;
            front = front + 1;
            return obj;
        }
    }
```

```
    public boolean isFull()
    {
    return (rear == (capacity - 1));
    }

    public boolean isEmpty()
    {
        return (front > rear);
    }

    public void print()
    {
        for ( int i = front; i <= rear; i++)
        {
            System.out.println("Data["+1+"] == > " + data[i]);
        }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل:

```
public class UseQueueByArray
{
    public static void main(String[] args)
    {
        QueueByArray queue = new QueueByArray(5);

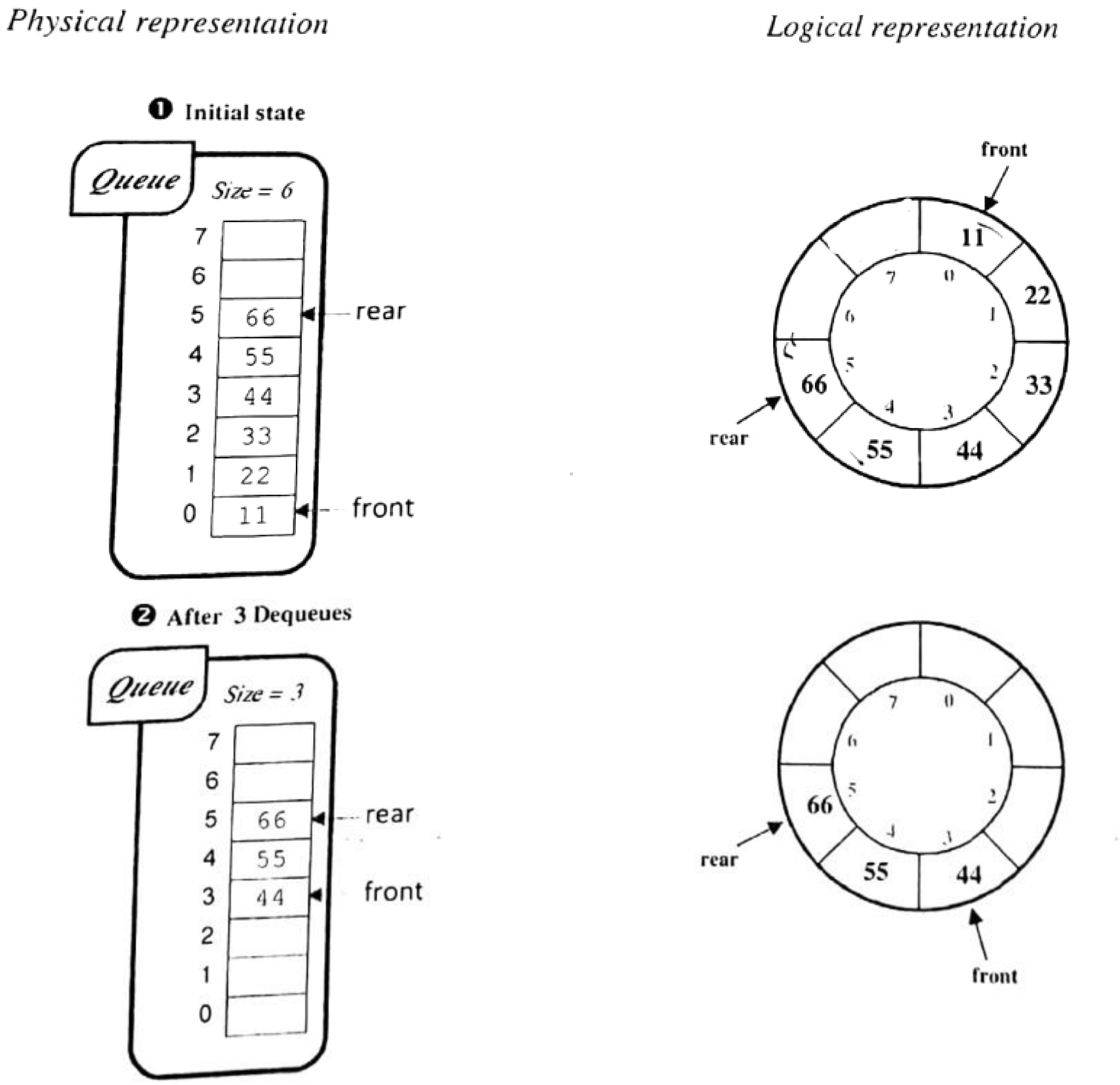
        System.out.println("\f");
        try
        {
            queue.enQueue(11);
            queue.enQueue(22);
            queue.enQueue(33);
            queue.enQueue(44);
            queue.enQueue(55);
            // queue.enQueue(66);

            queue.print();
            System.out.println("\n***************************
            ********\n");
```

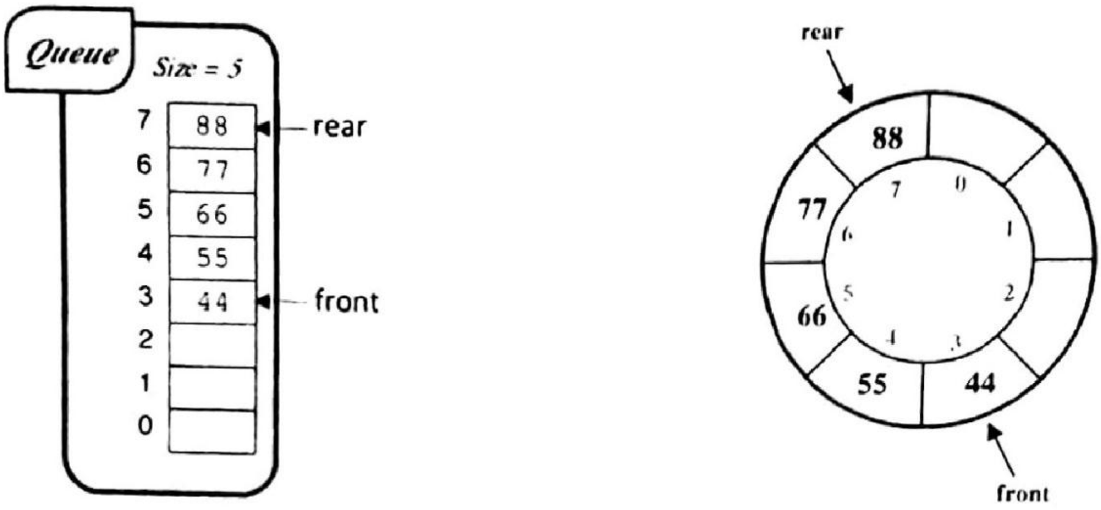
```
            System.out.println(queue.deQueue() + " has been
            removed ");
            System.out.println(queue.deQueue() + " has been
            removed ");
            System.out.println(queue.deQueue() + " has been
            removed ");
            System.out.println(queue.deQueue() + " has been
            removed ");
            System.out.println(queue.deQueue() + " has been
            removed ");
            System.out.println(queue.deQueue() + " has been
            removed ");
        }
        catch (Exception e)
        {
            System.out.println("ERROR" + e.getMessage());
        }
        queue.print();
    }
}
```

الطابور الدائري

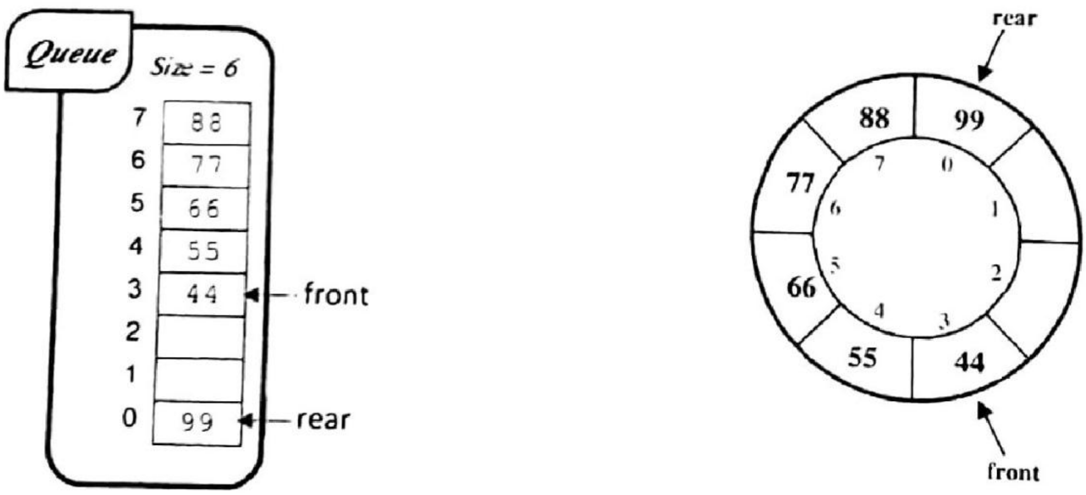
كما لاحظنا من الطابور الخطي فإن الطابور قد يكون ممتلئ بالرغم من وجود فراغات في بداية الطابور بعد عمليات الحذف dequeue، ولكي يتم الإستفادة من هذه الأماكن الفارغة التي تكونت في بداية الطابور قد يتم إزاحة كل عناصر الطابور لتحتل الأماكن الفارغة في البداية ومن ثم يصبح هناك فراغات في نهاية الطابور من جديد، لكن هذا الحل مكلف وليس عملي فلحل هذه المشكلة تم استخدام الطابور الدائري حيث تتم عملية الإضافة بزيادة 1 إلى مؤشر rear ووضع البيانات الجديدة في نهاية الطابور إلى أن يتم تعبئة المصفوفة بشكل كامل بالبيانات وفي حال ما إن أصبح rear مساويا (capacity - 1) ونريد إضافة عنصر جديد يتم قلب الطابور من أعلى إلى أسفل أي تصبح rea=1، وكذلك الحال بالنسبة لعمليات الحذف والمؤشر front. بمعني عند وصول مؤشر الـrear أو front إلى نهاية المصفوفة يتم قلبه إلى بداية المصفوفة، ويتم هذا بزيادة الـ rear /front بمقدار 1 ومن ثم إيجاد باقي القسمة على حجم المصفوفة. سنوضح بالأمثلة كيف تتم عمليات الإضافة (Enqueue) والحذف (Dequeue) في الطابور الدائري:



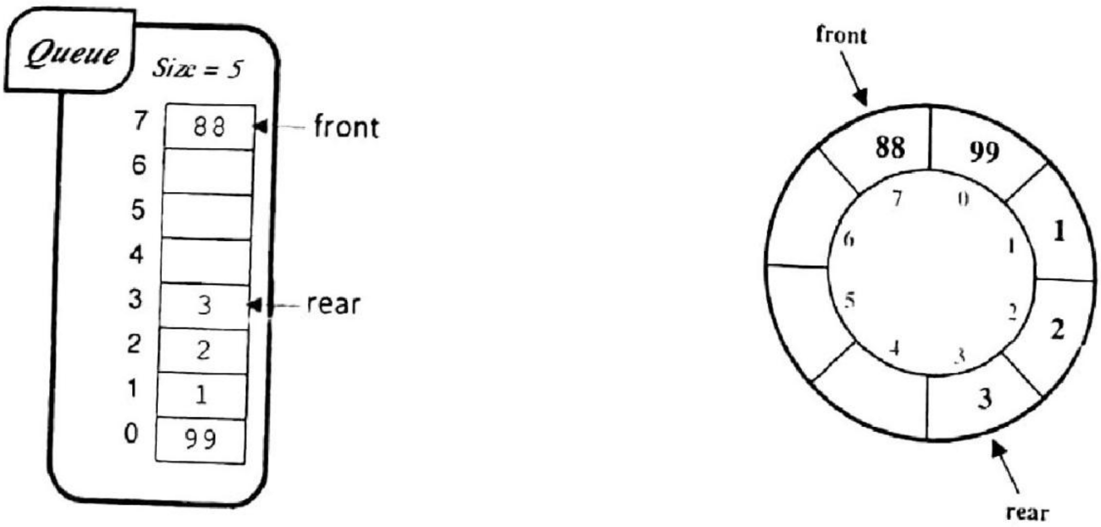
❸ **After 2 Enqueues**

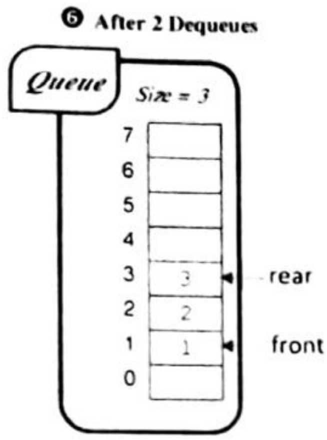


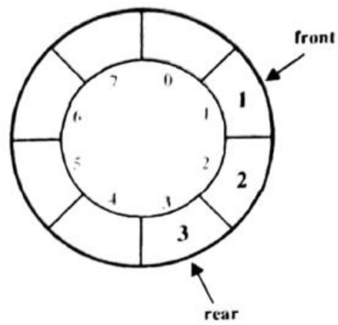
❹ **After Enqueue (99)**



❺ **4 Dequeues & 3 Enqueues**







نلاحظ من الأشكال السابقة أننا استخدمنا متغير جديد اسمه size، والذي يدل على حجم البيانات الفعلية المخزنة في الطابور وهو يختلف عن سعة الطابور الإجمالي capacity. شرط الإمتلاء هنا يتحقق عندما يكون الحجم الفعلي للبيانات بالطابور size مساويا لسعة الطابور الكلية capacity. باستخدام الطابور الدائري يمكننا استغلال كل الفراغات ضمن المصفوفة.

قبل القيام بإجراء العمليات المختلفة على الطابور الدائري باستخدام المصفوفات سنقوم أولا بتكوين فصيل جديد اسمه CircularQueue، يحتوي تعريف لمصفوفة من الكائنات (هنا سنستخدم الأعداد الصحيحة) اسمها data، ذات حجم أو سعة (capacity) يتم تحديدها ضمن بانيها، بالإضافة إلى ثلاثة حقول أخرى: واحد اسمه: front، ويدل على موقع أول عنصر في الطابور، والثاني اسمه: rear، ويدل على أخر عنصر في الطابور، والثالث اسمه: size، يدل على العدد الفعلي للبيانات المخزنة بالطابور وكما يلي:

```
public class CircularQueue
{
     private int front;
     private int rear;
     private int size;
     private int capacity;
     private Object[] data;
     public CircularQueue (int c)
     {
          capacity = c;
          data = new Object [capacity];
          front = 0;
          rear = capacity -1;
          size = 0;
     }
}
```

لاحظ أنه عند تكوين كائن جديد من فصيل الطابور الدائري فإن مؤشر الـfront سيكون عند 0 ومؤشر الـrear سيكون على (capacity -1)، ويتم تحديد سعة الطابور عن طريق القيمة c التي يتم استقبالها ضمن الباني، وأن الحجم الفعلي للبيانات **size** هو 0 لأن الطابور فارغ عند تكوينه.

الآن يمكننا استخدام هذا الفصيل ضمن فصيل آخر وليكن UseCircularQueue وتكوين كائن من الطابور الدائري كالتالي:

```
CircularQueue ca = new CircularQueue(5);
```

هنا حددنا سعة الطابور بـ5 عناصر كحد أقصى. الرقم 5 سيمرر إلى باني الفصيل CircularQueue ليتم تخزينه ضمن الحقل capacity ومن تم تحديد طول الطابور الدائري بـ5.

منهج إضافة عنصر إلى الطابور الدائري Enqueue

هذا المنهج سيقوم بإضافة عنصر جديد يتم استقباله كمعلمة إلى نهاية الطابور. في حال ما إن كان الطابور ممتلئ فسيتم إلقاء استثناء بذلك. مع كل عملية إضافة جديدة سيتم زيادة 1 إلى قيمة الحقل size، وأيضا تعديل قيمة الـ**rear** بإضافة 1 إليه ومن تم إيجاد باقي القسمة على الحجم الكلي للمصفوفة، هذا سيؤدي إلى جعل الـ**rear** ينتقل إلى بداية المصفوفة عند محاولة إضافة عنصر جديد والـrear عند الموقع (capacity - 1).

```
public void enQueue (Object object) throws Exception
{
     if (isFull())
          throw new Exception (" (2) : The queue is full !");
     else
     {
          size ++;
          rear = (rear + 1) % capacity;
          data [rear] = object;
     }
}
```

يمكننا استدعاء هذا المنهج الآن ضمن الفصيل UseCircularQueue عدة مرات لإضافة عناصر وكالتالي:

```
ca.enQueue(11);
ca.enQueue(22);
```

منهج هل الطابور ممتلئ isFull

هذا المنهج يختبر ما إن كان الطابور ممتلئ أم لا. شرط الإمتلاء يتحقق كما ذكرنا سابقا عندما يكون الحجم الفعلي للبيانات (أي قيمة الـsize) مساوية لـ5 أي حجم الطابور الكلي (capacity) والمنهج كالتالي:

```
public boolean isFull()
{
     return(size == capacity);
}
```

منهج هل الطابور فارغ isEmpty

يكون الطابور فارغا عندما يكون حجم البيانات الفعلي size مساويا للصفر، فالمنهج يكون كالتالي:

```
public boolean isEmpty()
{
     return size == 0;
}
```

منهج حذف عنصر من الطابور Dequeue

تتم عملية حذف العنصر الموجود عند مقدمة الطابور. هذا المنهج سيعيد عنصر البيانات المراد حذفه. لاحظ أنه سيتم زيادة 1 إلى الـ front ومن ثم إيجاد باقي القسمة على سعة المصفوفة الكلية capacity لتكون القيمة الجديدة هي قيمة الـfront هذا سيؤدي لانتقال الـfront من نهاية الطابور إلى بدايته عند محاولة حذف عنصر جديد والـfront يكون عند نهاية المصفوفة. سيتم إلقاء استثناء في حال ما إن كان الطابور فارغاً. لاحظ أيضا أنه يتم إنقاص 1 من حجم الطابور الفعلي size مع كل عملية حذف.

```
public Object deQueue() throws Exception
{
if (isEmpty())
     throw new Exception (" (1): The queue is empty !");
     else
     {
     Object obj = data[front];
     data[front] = null;
     front = (front + 1) % capacity;
     size--;
```

```
    return obj;
    }
}
```

يمكنك استدعاء هذا المنهج من الفصيل UseCircularQueue كالتالي:

```
System.out.println(ca.deQueue() + " at the front has been removed");
```

منهج زيارة "طباعة" عناصر الطابور الدائري print

يمكننا المرور أو زيارة كل عنصر من عناصر الطابور الدائري وذلك لإجراء أية عملية معالجة أخرى أو لطباعته. يتم المرور على كل عنصر في الطابور باستخدام حلقة for ابتداءً من الـfront إلى أن نصل إلى العنصر الذي يقع قبل الـ rear. لاحظ أن مقدار الزيادة i يتم بإضافة 1 إليه، ثم إيجاد باقي القسمة على حجم المصفوفة الكلي capacity، لأن في الطابور الدائري قد يكون موقع الـrear سابق لموقع الـfront. لاحظ أيضا أن أخر عنصر بالطابور الدائري (أي العنصر الواقع عند الموقع rear) يتم طباعته لوحده خارج الحلقة، لأن آخر عنصر بالحلقة هو ما قبل الـrear.

```
public void print()
{
    if (!isEmpty())
    {
        for (int i = front; i != rear; i = (i+1)%capacity)
        {
            System.out.println("Data["+i+"] ==> " + data[i]);
        }
        System.out.println("Data["+rear+"]    ==>        "    +
        data[rear]);
    }
}
```

يمكن استدعاء منهج الطباعة السابق كالتالي:

```
ca.print();
```

التالي هو البرنامج كاملا محتويا على كل المناهج سالفة الذكر.

```
public class CircularQueue
{
    private int front;
    private int rear;
    private int size;
    private int capacity;
    private Object[] data;

    public CircularQueue (int c)
    {
        capacity = c;
        data = new Object [capacity];
        front = 0;
        rear = capacity -1;
        size = 0;
    }

    public void enQueue (Object object) throws Exception
    {
        if (isFull())
            throw new Exception (" (2) : The queue is full !");
        else
        {
            size ++;
            rear = (rear + 1) % capacity;
            data [rear] = object;
        }
    }

    public Object deQueue() throws Exception
    {
        if (isEmpty())
            throw new Exception (" (1): The queue is empty !");
        else
        {
            Object obj = data[front];
            data[front] = null;
            front = (front + 1) % capacity;
            size--;
            return obj;
        }
    }
```

```
    public int getSize()
    {
         return size;
    }

    public boolean isFull()
    {
         return(size == capacity);
    }

    public boolean isEmpty()
    {
         return size == 0;
    }

    public void print()
    {
         if (!isEmpty())
         {
              for (int i = front; i != rear; i = (i+1)%capacity)
              {
                   System.out.println("Data["+i+"]    ==>    "    +
              data[i]);
              }
              System.out.println("Data["+rear+"]   ==>      "   +
              data[rear]);
         }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل:

```
public class UseCircularQueue
{
    public static void main(String[] args)
    {
        CircularQueue ca = new CircularQueue(5);

        System.out.println("\f");
        try
        {
```

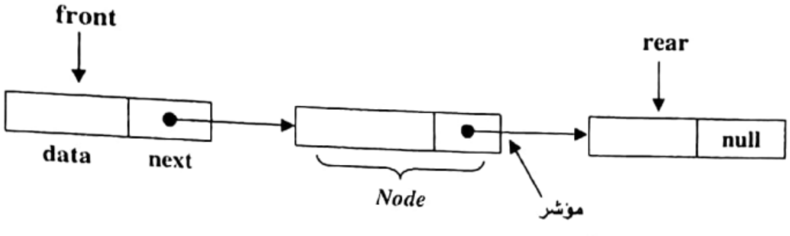
```
            ca.enQueue(10);
            ca.enQueue(22);
            ca.enQueue(30);
            ca.enQueue(40);
            ca.enQueue(50);
            // ca.enQueue(60);

            ca.print();
            System.out.println("\n********************\n");

            System.out.println(ca.deQueue()  +  "  has  been
        removed");
            System.out.println(ca.deQueue()  +  "  has  been
        removed");
            System.out.println(ca.deQueue()  +  "  has  been
        removed");
            System.out.println(ca.deQueue()  +  "  has  been
        removed");
            System.out.println(ca.deQueue()  +  "  has  been
        removed");
        }
        catch (Exception e)
        {
            System.out.println("ERROR" + e.getMessage());
        }
        ca.print();
    }
}
```

ثانيا تمثيل الطابور باستخدام القوائم المترابطة

الطابور باستخدام القائمة المترابطة: هي عبارة عن مجموعة من العقد، كل عقدة تتكون من جزئين: الجزء الأول data، يتم فيه تخزين البيانات، والجزء الثاني next فهو مؤشر أو مرجع للعقدة التالية في الطابور. هناك أيضا مؤشران، الأول يشير إلى مقدمة الطابور front ومؤشر أخر يشير إلى آخر عقدة في الطابور rear.



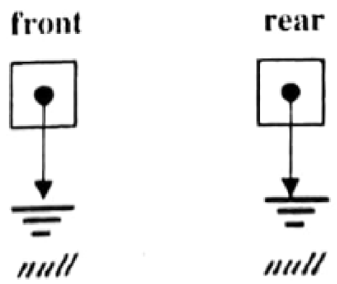
لاحظ أن فصيل العقدة الذي سنستخدمه في الطابور هو نفس فصيل العقدة الذي استخدمناه في القائمة المترابطة والمكدس، لذلك سوف لن نعيد برمجته مرة أخرى. قبل القيام بإجراء العمليات المختلفة على القائمة المترابطة (الطابور) سنقوم أولا بتكوين فصيل جديد اسمه: QueueByLinkedList يحتوي على حقلين، واحد يؤشر لمقدمة الطابور front والآخر يؤشر إلى أخر عنصر بالطابور rear وكما يلي:

```
public class QueueByLikedList
{
     protected Node front;
     protected Node rear;

     public QueueByLinkedList()
     {
          front = null;
          rear = null;
     }
}
```

الآن يمكننا استخدام هذا الفصيل ضمن فصيل آخر وليكن UseQueue، وتكوين كائن من الطابور كالتالي:

لاحظ أنه عند تكوين كائن جديد من فصيل الطابور فإن الطابور سيكون فارغا، وأن كل من مؤشري المقدمة front والمؤخرة rear تساوي الـnull كما بالشكل التالي:



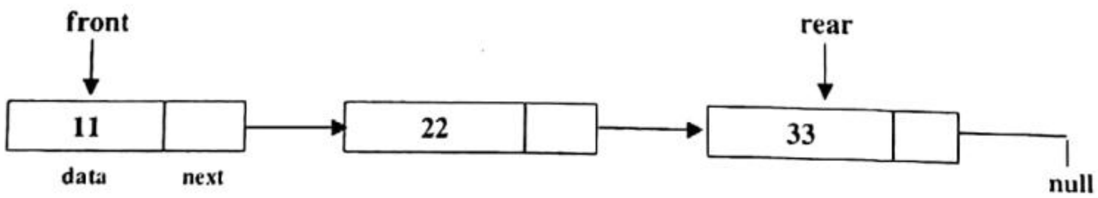
منهج إضافة عقدة إلى الطابور Enqueue

هذا المنهج يقوم بإضافة عقدة جديدة إلى أخر الطابور بحيث يخزن جزء البيانات (data) القيمة الجديدة المراد إضافتها، وتخصيص null لجزء الـnext، لأنها ستكون آخر عقدة بالقائمة المترابطة. ثم يتم تغيير مؤشر الـnext للعقدة الأخيرة في الطابور ليؤشر إلى العقدة الجديدة مع الأخذ في الإعتبار أنه إذا كان الطابور فارغا فإن العقدة الجديدة ستكون هي العقدة الوحيدة في الطابور وأن مؤشر المقدمة front يجب أن يشير إلى العقدة الجديدة. ولكون العقدة الجديدة تمثل آخر عقدة في الطابور فيجب أيضا تعديل مؤشر المؤخرة rear ليشير إلى هذه العقدة الجديدة، وكما هو موضح بالمنهج التالي:

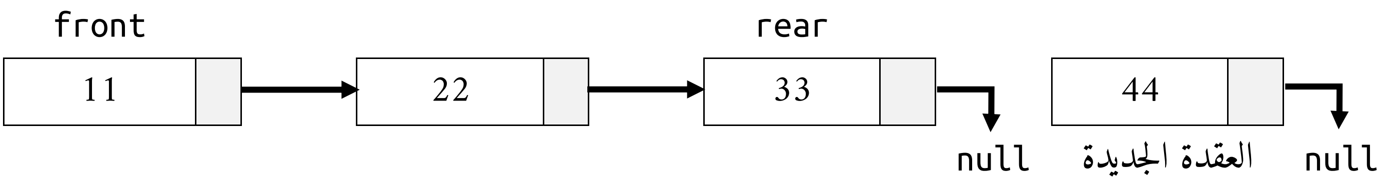
```
public void enqueue (Object item)
{
     Node newNode = new Node (item, null);
     if (isEmpty())
          front = newNode;
     else
          rear.next = newNode;
     rear = newNode;
}
```

والمثال التالي يوضح هذه العملية:

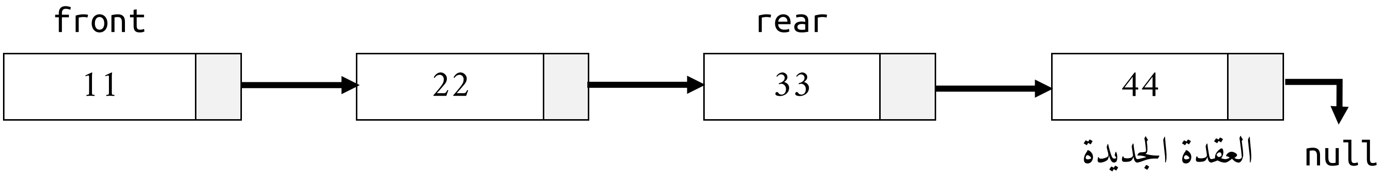
لنفترض أن لدينا الطابور التالي ونريد إضافة عقدة جديدة عند آخره.



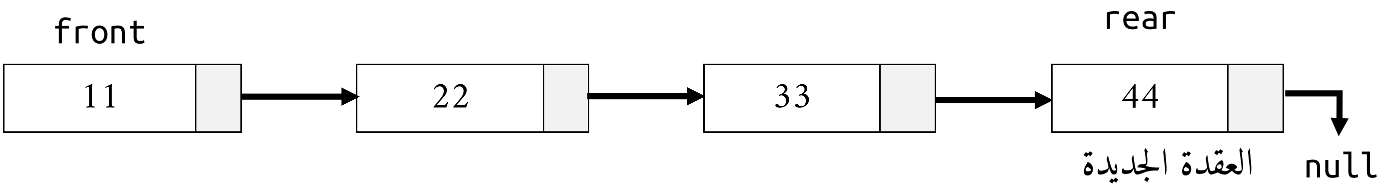
1- ننشئ عقدة جديدة (newNode) وندخل بيانات العقدة الجديدة (لتكن مثلا: 44)، وكذلك المؤشر (وهو هنا: null).



2- نحدث جزء الـnext لاخر عقدة ليؤشر إلى العقدة الجديدة (newNode).



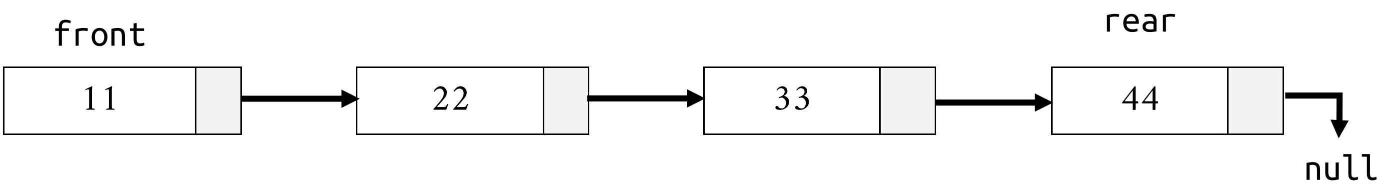
نحدث مؤشر الـrear ليشير إلى العقدة الجديدة (newNode).



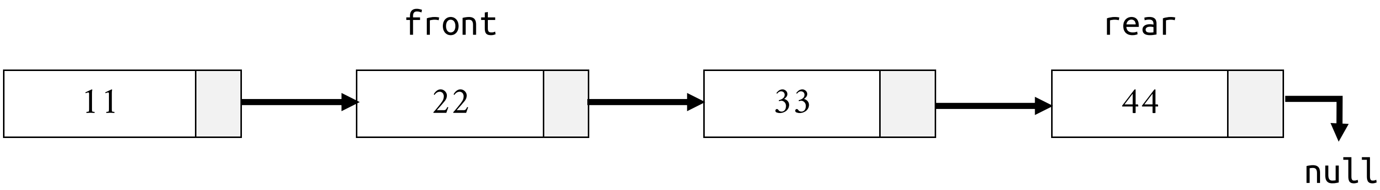
حذف عقدة من الطابور Dequeue

عملية حذف عقدة من الطابور تتم عند مقدمة الطابور (أي أول عنصر تم إدخاله سيتم حذفه أولا). تتم عملية الحذف بجعل مؤشر المقدمة front يشير إلى العقدة التالية (أي يؤشر إلى ما يؤشر إليه جزء الـnext للعقدة الواقعة في مقدمة الطابور). لاحظ أن منهج الـ**dequeue** سيقوم بإرجاع جزء البيانات data من العقدة الواقعة في مقدمة الطابور قبل حذفها. أما بالنسبة للعقدة القديمة والتي كانت في مقدمة الطابور فإن المنظف في الجافا ***garbage Collector*** سوف يتخلص منها ويعيدها للذاكرة مرة أخرى، فلا داعي لفعل أي شيء بالنسبة لها. فقط هناك استثناء سيتم إلقاؤه في حال محاولة الحذف من الطابور الفارغ. أيضا يجب الأخذ في الإعتبار أنه عندما يكون هناك عقدة واحدة فقط في الطابور فإن عملية حذف هذه العقدة سيؤدي لجعل كل من مؤشري المقدمة front و المؤخرة rear يساوي null، والمثال التالي يوضح عملية حذف عقدة من الطابور:

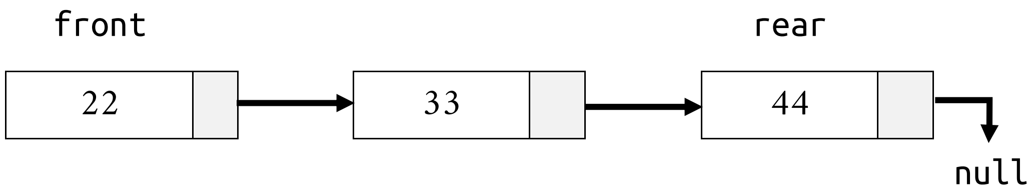
لنفترض أن لدينا الطابور التالي، ونريد القيام بعملية الحذف عليه()dequeue.



1- نغير مؤشر الـfront ليشير إلى العقدة التالية (أي إلى ما يشير إليه جزء الـnext للعقدة الواقعة في مقدمة الطابور (بمعنى front.next)) كالتالي:



2- الطابور سيصبح كالتالي بعد أن يقوم الـ ***garbage collector*** في الجافا بحذف العقدة القديمة:



والمنهج التالي يقوم بهذه العملية:

```
public Object dequeue() throws Exception
{
     Object obj;
     if (isEmpty())
          throw new Exception("The Queue is empty !!");
     else
     {
          obj = front.data;

          if (front == rear)
               rear = rear.next;

          front = front.next;
     }
     return obj;
}
```

لاحظ أن معظم المناهج التي قمنا بكتابتها في القائمة المترابطة سابقا يمكن تطبيق هنا كما هي وبدون تغيير، ولذلك فلا داعي لتكرارها من جديد، من هذه المناهج مثلا:

- المنهج هل القائمة المترابطة فارغة isEmpty().
- منهج الطباعة وزيارة عقد القائمة بشكل تتابعي print().

التالي هو البرنامج كاملا محتويا على كل المناهج السابقة.

```
public class QueueByLikedList
{
    protected Node front;
    protected Node rear;

    public QueueByLinkedList()
    {
        front = null;
        rear = null;
    }

    public void enqueue (Object item)
    {
        Node newNode = new Node (item, null);

        if (isEmpty())
            front = newNode;
        else
            rear.next = newNode;

        rear = newNode;
    }

    public Object dequeue() throws Exception
    {
        Object obj;

        if (isEmpty())
            throw new Exception("The Queue is empty !!");
        else
        {
            obj = front.data;

            if (front == rear)
                rear = rear.next;

            front = front.next;
        }
        return obj;
```

```
    }

    public boolean isEmpty()
    {
        return (front == null);
    }

    public Node getFront()
    {
        return front;
    }

    public Node getRear()
    {
        return rear;
    }

    public void print()
    {
        for (Node p = front; p != null; p = p.next)
        {
            System.out.println( "Data : " + p.data);
        }
    }
}
```

البرنامج التالي يقوم باختبار هذا الفصيل:

```
public class UseQueue
{
    public static void main(String[] args)
    {

        System.out.print("\f");
        try
        {
            QueueByLinkedList queue = new QueueByLinkedList();

            queue.enqueue(10);
            queue.enqueue(20);
```

```
            queue.enqueue(30);
            queue.enqueue(40);
            queue.enqueue(50);
            queue.enqueue(60);

            queue.print();
            System.out.println("\n*********************\n");

            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            System.out.println(queue.dequeue()  +  "  has  been
        removed");
            queue.print();
        }
        catch (Exception e)
        {
           System.out.println(e.getMessage());
        }
    }
}
```

طابور الأولوية priority queue

في هذا الطابور يتم تجميع المهام المراد القيام بها في أولويات، بحيث يتم وضع المهام ذات الأهمية الأكبر في بداية الطابور والمهام ذات الأهمية الأدني في نهاية الطابور. وبالتالي يتم تنفيذ الأولويات ذات المستوى الأعلى أولا لوقوعها في بداية الطابور. يتم تمثيل طابور الأولوية عادة باستخدام القوائم المترابطة بحيث يتم ربط كل مجموعة من العناصر ذات نفس الأولوية في طابور فرعي له مؤشري مقدمة ومؤخرة خاص به. وعند إجراء عملية إضافة عنصر يتم تحديد أولويته أولا ومن ثم يضاف إلى نهاية الطابور الفرعي الخاص بتلك الأولوية. أما عملية الحذف فتبدأ من الطابور الجزئي الأكثر أولوية

أولا، ويتم حذف كل العناصر بهذا الطابور الجزئي الواحد تلو الأخر، ثم ينتقل الإلغاء إلى الطابور الجزئي التالي له في الأولوية وهكذا. كمثال لذلك لنفترض أن لدينا مؤسسة ما تقوم بتنفيذ مهام الطباعة على طابعة مشتركة بين كل الموظفين ونريد أن نضع أولويات للطباعة فيما يخص الموظفين على حسب الدرجة الوظيفية، كأن يتم تنفيذ مهام طباعة المدارء قبل رؤساء الأقسام، وتنفيذ مهام طباعة رؤساء الأقسام قبل الموظفين العاديين. هنا يتم تكوين طابور أولوية بحيث يتم إضافة مهام الطباعة القادمة من المدراء في طابور جزئي ذات أهمية عليا وليكن (A) في بداية الطابور، ومن ثم إضافة مهام طباعة رؤساء الأقسام والتي تليه في الأولوية في طابور جزئي ذات أهمية متوسطة وليكن(B)، ومن ثم إضافة مهام الطباعة القادمة من الموظفين العاديين في طابور جزئي ثالث وليكن (C) في نهاية الطابور. هنا يتم تنفيذ مهام الطباعة في الطابور الجزئي A أولا إلى أن يتم إكمال كل مهام الطباعة القادمة من المدراء، بعد ذلك يتم تنفيذ كل مهام الطباعة في الطابور B القادمة من رؤساء الأقسام، وبعد ذلك يتم تنفيذ مهام الطباعة بالنسبة للطابور C.

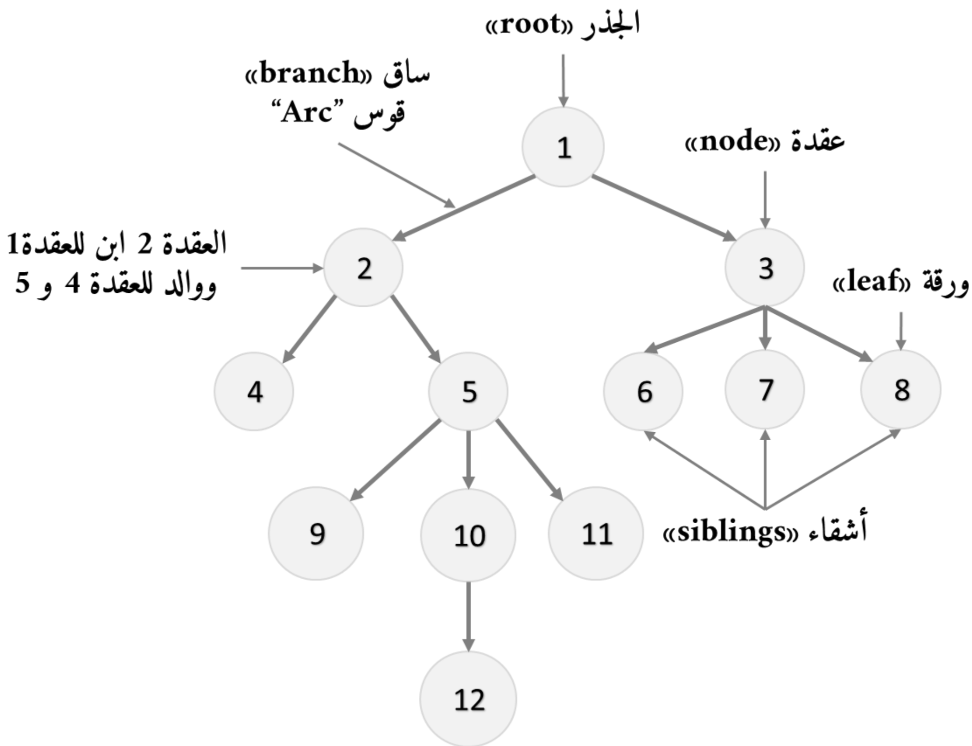
تطبيقات الطابور

بشكل عام فإن للطابور تطبيقات متعددة في حياتنا العملية وكذلك أنظمة التشغيل والشبكات فمنها على سبيل المثال:

- ❖ مهام الطباعة المختلفة التي يتم إرسالها إلى طابعة مشتركة بين عدة حواسيب أو حتى عدة مهام طباعة يتم تنفيذها على حاسوب واحد، كل ذلك يتطلب استخدام الطابور لتنظيم عملية الطباعة.
- ❖ أنظمة التشغيل المتنوعة تستخدم الطوابير بشكل أساسي لمعالجة المهام المختلفة.
- ❖ شبكات الحاسوب تستخدم الطابور لتنظيم طالبات المستخدمين.
- ❖ الأنظمة المستخدمة في عدة محلات وشركات والتي تقوم بتنظيم وصول الزبائن، حيث أن كل زبون يستلم عند وصوله قسيمة الإنتظار (ويدرج ضمن طابور الإنتظار)، ثم يجلس الزبون إلى أن يحين دوره ويظهر رقمه على شاشة العرض ليتم تقديم الخدمة له.
- ❖ أنظمة الإنتظار في العيادات والمستشفيات، تستخدم الطابور لتنظيم الخدمة المقدمة للمرضى والمراجعين.
- ❖

الشجرة Tree

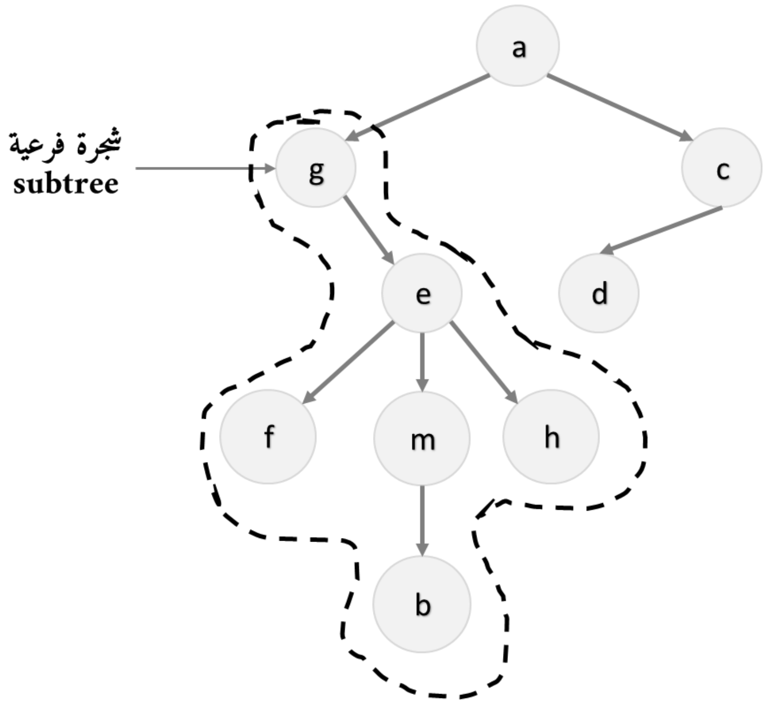
ما تم شرح حتى الأن من مصفوفات وقوائم مترابطة ومكدس هي هيا كل بيانات خطية، ولكن الشجرة تعتبر من ضمن هيا كل البيانات الغير خطية والتي يتم تمثيل البيانات فيها بصورة هرمية. فمفهوم الشجرة ضمن هيا كل البيانات يأخذ الوضع المقلوب رأسا على عقب لما هو موجود بالشجرة في الحياة الطبيعية. فالشجرة هي هيكل بيانات يتكون من عقد وسيقان حيث أن العقدة في أعلى الشجرة تسمى الجذر "root" وأن الأغصان "branches" والأوراق "leaves" تكون في أسفل الهيكل، كما هو موضح بالشكل التالي:



في هيكل الشجرة كل عقدة (عدا الجذر) يسبقها عقدة واحدة فقط، ويمكن أن يليها صفرا أو أكثر من العقد. إن لم تكن العقدة جذرا أو ورقة فيسبقها عقدة واحدة فقط تسمى الوالد/الأب "parent" ويليها واحدة أو أكثر من العقد، تسمى الأبناء: children.

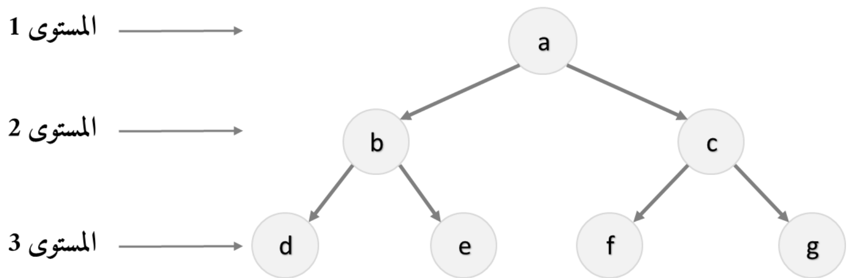
لاحظ أن معظم المصطلحات المستخدمة في هيكل الشجرة هي مستمدة أصلا من الشجرة الحقيقة أو شجرة العائلة. فشجرة العائلة يمكن تمثيلها بهيكل الشجرة حيث يكون الجد في عقدة الجذر والأباء في العقد التالية والأبناء في العقد التالية للأباء والأحفاد في عقد الأوراق.

تسمى العقد التي تشترك في نفس الوالد بالأشقاء "siblings". يتم تمثيل البيانات بواسطة العقد، وفي حالة إن كانت الشجرة لا تحتوي على عقد فتسمى الشجرة الفارغة "***Empty tree***". عادة ما تحتوي العقدة بالإضافة إلى البيانات على مؤشرات إلى عقد الأبناء التالية لها؛ نظرا لأن كل عقدة هي أب للعقد التالية لها، فإن المسار الذي يتبع سلسلة من العقد ابتداء من العقدة a إلى العقدة b من ضمن سلالتها تسمى مسارا. يطلق على العقدة b سليل "***ancestor***" العقدة a، ويطلق على العقدة a سلف "***descendent***" العقدة b. ويطلق على العقدة g مثلا مع كل سلالتها من العقد بالشجرة الفرعية "***subtree***" كما موضح بالشكل التالي:



لاحظ أن كل عقدة ضمن الشجرة تقع في مستوى معين، فمثلا: في المثال التالي: عقدة a تقع في المستوى الأول (1) والعقدة b و c يقعان في المستوى الثاني (2) و d في المستوى الثالث (3) وهكذا، وهذا ما يسمى بعمق العقدة، أو مستوى العقدة في الشجرة، ويحسب كالتالي:

مستوى العقدة في الشجرة = عدد العقد السابقة له + 1



وأن ارتفاع أو عمق الشجرة فهو أعلى مستوى في الشجرة، في المثال السابق يساوي 3.

ملاحظة:

بعض المراجع تصف مستوى الجذر بالمستوى 0 وليس 1 والمستوى التالي له بمستوى 1 وليس 2 وهكذا.

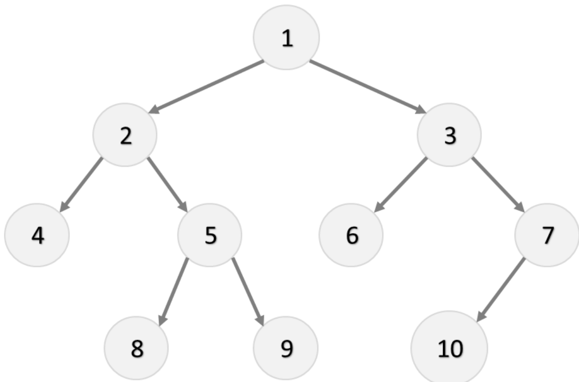
ما الفائدة من الأشجار؟

- ❖ أهم ميزة لاستخدام الأشجار في تمثيل المعلومات بصورتها الهرمية الطبيعية، مثلا: تمثل نظام الملفات و الأدلة (المجلدات) في الحاسوب، وتمثيل شجرة العائلة ... إلخ).
- ❖ تمنح الأشجار بطريقة ترتيب معينة (شجرة البحث الثنائية BST) طريقة بحث ووصول مناسبة (فهي أسرع من القوائم المترابطة وأبطئ من المصفوفات في عمليات البحث والوصول إلى البيانات).
- ❖ تعطي الأشجار طريقة مناسبة للإضافة والحذف (فهي أسرع من المصفوفات؛ ولكنها أبطى من القوائم المترابطة في عمليات الإضافة و الحذف).
- ❖ ليس للأشجار حد أقصى من العقد مثلها مثل القوائم المترابطة وعلى عكس المصفوفات، وذلك حين تمثل الأشجار بالقوائم المترابطة.

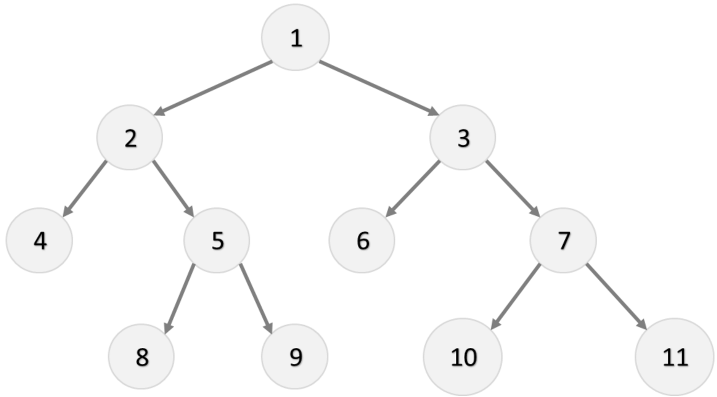
عادة ما تكون للشجرة رتبة معينة، فإذا احتوت كل عقدة في الشجرة على طفلان على الأكثر، تسمى بـ: الشجرة الثنائية Binary free وإذا احتوت على ثلاثة أطفال على الأكثر في كل عقدة تسمى ثلاثية "ternary tree" وهكذا. لذلك سنقوم بدراسة الأشجار الثنائية بشئ من التفصيل لما لها من استخدامات متعددة وسهولة تطبيقها.

الشجرة الثنائية "Binary tree"

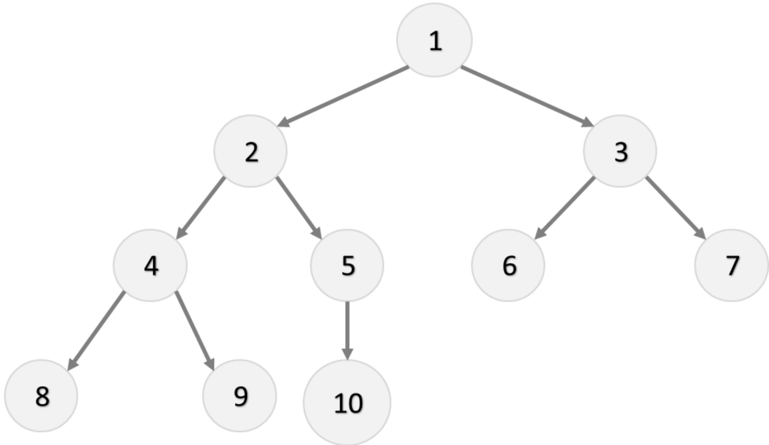
هي التي يملك جميع عناصرها على الأكثر ولدان، أحدهما يسمى الطفل الأيسر "left"، والثاني يسمى الطفل الأيمن "right"، كما بالشكل التالي:



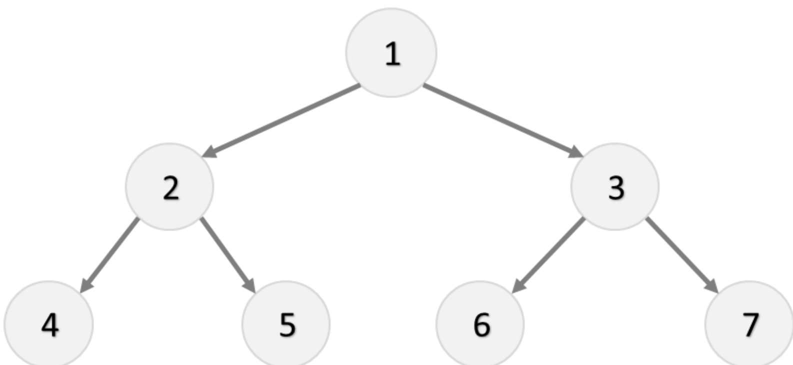
للشجرة الثنائية عدة أشكال، فمنها الشجرة الثنائية الممتلئة "Full binary tree" حيث يكون لكل العقد الداخلية أطفال عدا الأوراق ليس لها أي أطفال، أو بمعنى آخر كل عقدة لها 0 أو 2 من الأطفال كما بالشكل التالي:



أما الشجرة الثنائية الكاملة "Complete binary tree" فكل عقد المستويات لها طفلين وتكون معبأ بشكل كامل، عدا المستوى الأخير والتي قد يكون ليس كامل، ولكن معبأ من اليسار إلى اليمين كما بالشكل التالي:



أما الشجرة التامة/المثالية "Perfect binary tree" كل العقد الداخلية لها طفلان، وكل المستويات تكون في نفس المستوى كما موضح بالشكل التالي:



بشكل عام فإن عدد العقد في الشجرة الثنائية ذات الإرتفاع h يمكن حسابه كالتالي:

المستوى الأول (1) = 2^0 = 1 عقدة

المستوى الثاني (2) =2^1 = 2 عقدة

المستوى الثالث (3) = 2^2 = 4 عقد

المستوى الرابع (4) = 2^3 = 8 عقد

.

.

.

المستوى الأخير (h) = 2^{h-1} من العقد

فمجموعة كل العقد في كل المستويات في الشجرة الثنائية = $\sum_{i=1}^{h} 2^{i-1}$ = $2^h - 1$

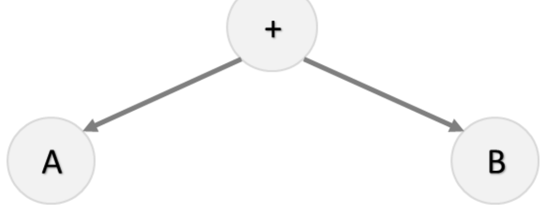
تطبيقات الأشجار

1- التعابير الرياضية

يمكن تمثيل التعابير الحسابية باستخدام الأشجار حيث يتم تمثيل العمليات الحسابية "operators" في العقد الداخلية وتمثيل المتغيرات أو الثوابت "Opercards" كأوراق، فمثلا التعبير الحسابي التالي:

A + B

يمكن تمثيله بالشجرة كما يلي:



في هذه الحالة يمكننا عبور الشجرة بواحدة من ثلاث طرق:

❖ العبور الوسطي inorder traversal (left-root-right)

يتم هذا العبور من خلال العقد (يسار-جذر-يمين) أي يتم زيارة الابن الأيسر أولا، ثم الجذر، ثم الابن الأيمن، وهذه الطريقة تعطينا صيغة التعبير في الـinfix:

A + B

❖ العبور القبلي preorder traversal (root-left–right)

يتم هذا العبور من خلال العقد (جذر-يسار-يمين) أي يتم زيارة الجذر أولا ثم الابن الأيسر ثم الابن الأيمن، وهذه الطريقة تعطينا صيغة التعبير في الـprefix:

+AB

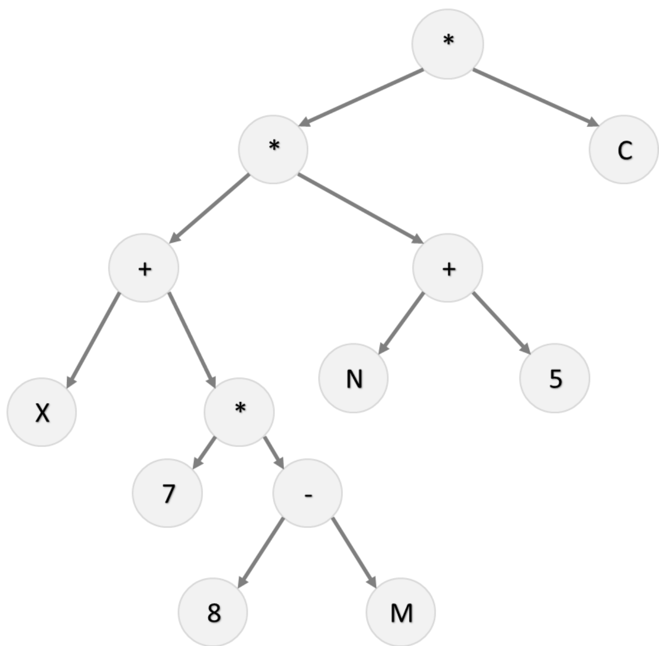
❖ العبور البعدي postorder traversal (left-right-root)

يتم هذا العبور من خلال العقد (يسار-يمين-جذر) أي يتم زيارة الابن الأيسر أولا، ثم الابن الأيمن ثم الجذر، وهذه الطريقة تعطينا صيغة التعبير في الـpostfix:

AB+

مثال: مثل التعبير الرياضي التالي في شجرة، وأوجد صيغة التعبير في كل من: 1-prefix 2-postfix

((X+7 * (8-M)) * (N+5))*C

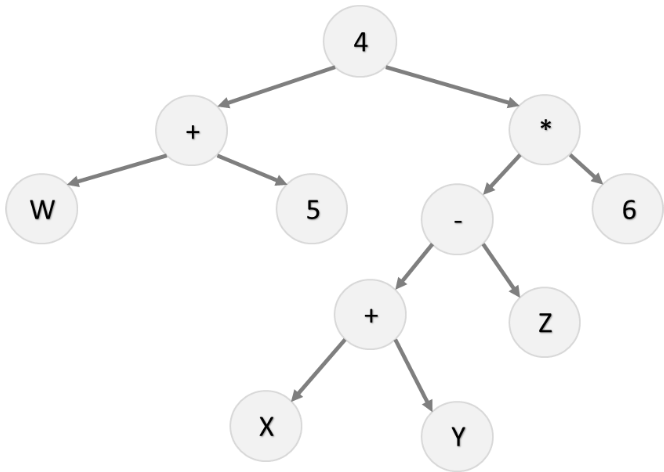


Prefix (preorder traversal): **+X*7-8M+N5C

postfix (postorder traversal) : X78M-*+N5+*C*

مثال: مثّل التعبير الرياضي التالي في شجرة وأوجد صيغة التعبير في كل من: prefix-1   postfix-2

(W+5)/((X+Y-Z)*6)

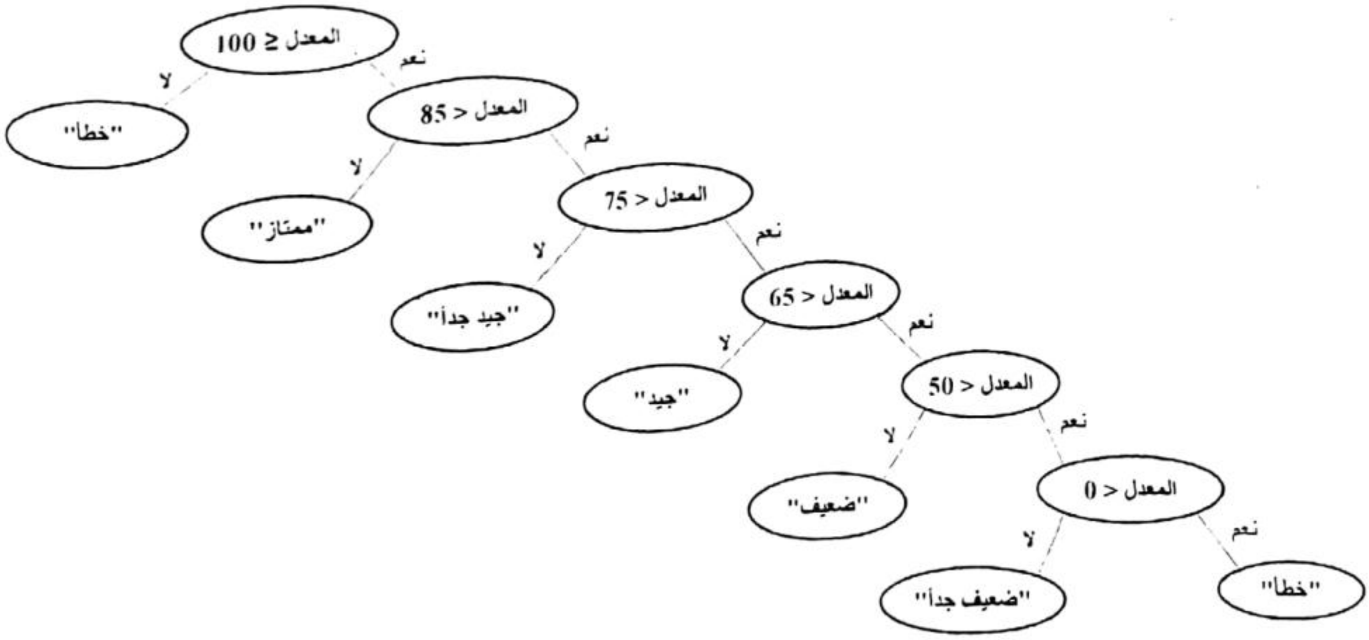


Prefix (preorder traversal): /+W5*-+XYZ6

postfix (postorder traversal): W5+XY+Z-6*/

2- شجرة القرارات Decision tree

يدخل جدول القرارات في العديد من التطبيقات الإدارية والهندسية ويعتبر من الجداول المهمة في تحليل النظم. تستخدم الشجرة في تمثيل شجرة القرارات وذلك بجعل العقد الداخلية تحتوي على أسئلة تكون إجابتها إما بـ"نعم" أو "لا"، أي أن العقد تمثل شرطا يترتب عليه قرار والذي يكون عادة ممثلا بالأوراق (أي المعالجة أو الإجراء المطلوب القيام به). الشجرة التالية تمثل نموذج بسيط لشجرة القرار:

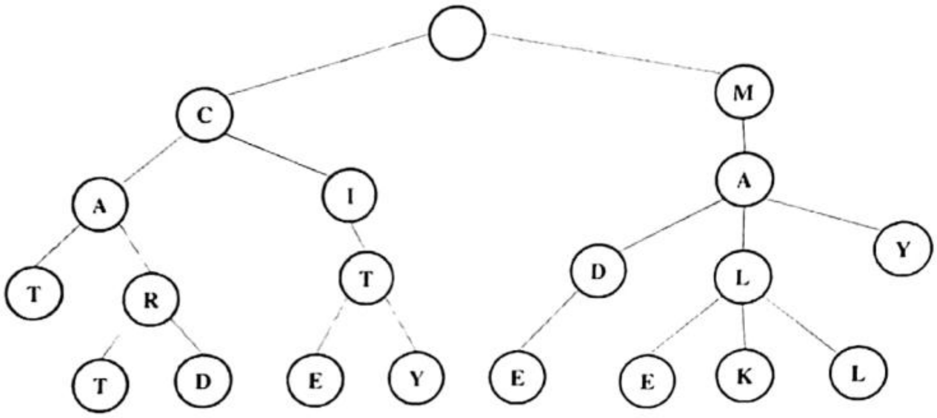


3- القواميس

لنفترض أن لديك الكلمات التالية:

CAT, CAR, CART, CARD, CITE, CITY, MAD, MADE, MALE, MALK, MALL, MAY

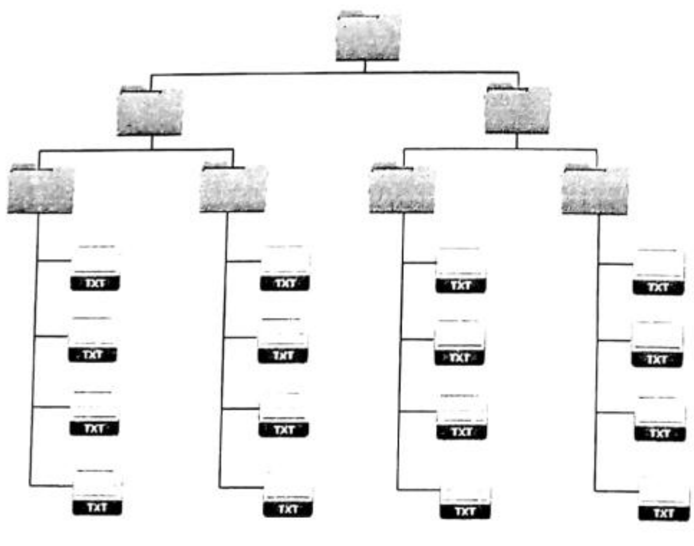
يمكننا استخدام الشجرة بشكلها العام لتمثيل هذه الكلمات كالتالي:



لاحظ أن عدد الحروف الأصلية للكلمات هي 44 حرف، وقد تم تمثيلها فقط بـ 19 حرف في الشجرة، مما وفر مساحة تخزين أكثر من النصف، إضافة إلى سرعة البحث والوصول إلى الكلمة.

4- تمثيل الأدلة (المجلدات) والملفات في أنظمة التشغيل

يكثر استخدام الشجرة بشكلها العام في هيكلة أنظمة التشغيل ولغات البرمجة المختلفة بحيث يتم تنظيم الأدلة والملفات في نظام إدارة الملفات في هيكل الشجرة، فالمثال التالي يوضح نموذج بسيط لهذا الاستخدام:



تمثيل الشجرة الثنائية:

عادة ما يتم تمثيل الأشجار باستخدام المصفوفات أو القوائم المترابطة، غير أن استخدام المصفوفات يحمل معه العديد من العيوب التي تجعل من استخدام القوائم المترابطة الأكثر تفضيلا لتمثيلها. فمن عيوب استخدام المصفوفات أن عمليتي الإضافة والحذف تؤدي إلى تغيير مواقع العناصر وهذا مكلف بالنسبة للمعالجة، بالإضافة إلى وجود العديد من الفراغات بالمصفوفات عادة، نظرا لعدم التوازن في تفرعات الشجرة مما يؤدي لعدم استغلال الذاكرة بشكل فعال. لذلك فيما يلي سنقوم بتمثيل الشجرة باستخدام القوائم المترابطة.

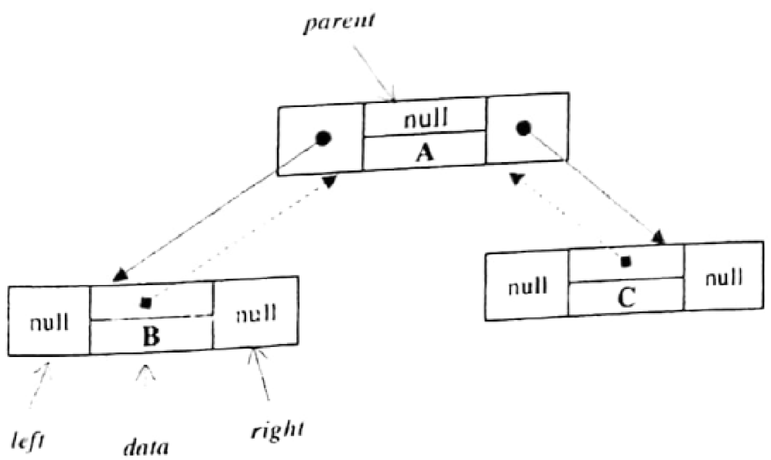
يتم تمثيل الشجرة في الجافا بمؤشر (reference) يشير إلى أعلى عقدة في الشجرة (الجذر). حيث تكون الشجرة فارغة عندما يكون مؤشر الجذر مساويا للـnull.

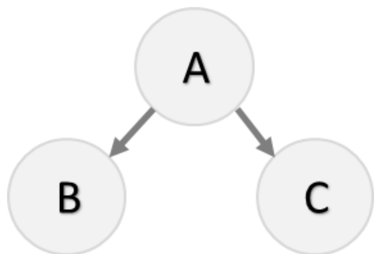
تتكون العقدة في الأشجار الثنائية عادة من ثلاثة أجزاء:

- جزء البيانات "data".
- مؤشر للطفل الأيمن "right".
- مؤشر للطفل الأيسر "left".

كما يمكننا إضافة مؤشر رابع يحمل قيمة الأب "parent" لتسهيل الوصول إلى الأب عند الحاجة، حيث أن التنقل ضمن الشجرة يتم عادة من الأعلى إلى الأسفل وليس العكس، فالإحتفاظ بمؤشر إلى الأب ضمن عقدة الابن سوف يسرع عملية الوصول إلى أب العقدة (في المستوى الأعلى منه).

يمكننا تمثيل الشجرة التالية باستخدام القوائم المترابطة كما يلي:





في الجافا يمكننا بناء فصيل (class) للعقدة Node كما يلي:

```
public class Node
{
    protected Node left;
    protected Object data;
    protected Node right;
    protected Node parent;

    public Node()
    {
        left = null;
        data = null;
        right = null;
        parent = null;
    }

    public Node(Object item)
    {
        left = null;
        data = item;
        right = null;
        parent = null;
    }

    public Node(Object item, Node parentPtr)
    {
        left = null;
        data = item;
        right = null;
        parent = parentPtr;
    }

    public Node(Node leftPtr, Object item, Node rightPtr, Node
parentPtr)
    {
        left = leftPtr;
        data = item;
        right = rightPtr;
        parent = parentPtr;
    }
```

```
    public void setData(Object item)
    {
        data = item;
    }

    public Object getData()
    {
        return data;
    }

    public void setLeft(Node leftPtr)
    {
        left = leftPtr;
    }

    public Node getLeft()
    {
        return left;
    }

    public void setRight(Node rightPtr)
    {
        right = rightPtr;
    }

    public Node getRight()
    {
        return right;
    }

    public void setParent(Node parentPtr)
    {
        parent = parentPtr;
    }

    public Node getParent()
    {
        return parent;
    }
}
```

كما يمكننا تكوين فصيل جديد اسمه BinaryTree ليمثل الشجرة الثنائية، حيث يحتوي على حقل واحد فقط وهو مؤشر للجذر "root" من نوع العقدة كما يلي:

```
public class BinaryTree
{
    public Node root;

    public BinaryTree()
    {
        root = null;
    }

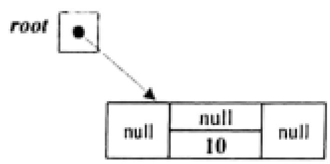
    public BinaryTree(Object item)
    {
        root = new Node(item);
    }
}
```

لاحظ أن هذا الفصيل يحتوي على بانيين: أحدهما لا يستقبل أي قيمة وتعطي null لمؤشر الجذر ليكون شجرة فارغة، والثاني يستقبل قيمة لعقدة الجذر، وبالتالي يكون شجرة تحتوي على عقدة واحدة هي عقدة الجذر.

الآن يمكننا تكوين كائن من فصيل الشجرة الثنائية كالتالي:

```
BinaryTree tree = new BinaryTree(10);
```

هنا سيتم تكوين شجرة ثنائية تحتوي على عقدة الجذر والتي تحتوي بياناته على القيمة 10 كالتالي:



منهج اختبار هل العقدة ورقة أم لا isLeaf

تعتبر العقدة ورقة عندما لا تملك أطفال من جهة اليمين ولا من جهة اليسار، كما هو موضح بالمنهج التالي:

```
public boolean isLeaf (Node n)
{
    return ( (n.left == null) && (n.right == null) );
}
```

منهج اختبار هل الشجرة فارغة isEmpty

تعتبر الشجرة فارغة عندما يشير مؤشر الجذر إلى null، كما هو موضح بالمنهج التالي:

```
public boolean isEmpty()
{
     return root == null;
}
```

منهج إضافة طفل أيسر addLeftChild

يقوم هذا المنهج بإضافة عنصر جديد كطفل أيسر لعقدة ما حيث يستقبل مؤشر للعقدة المراد إضافة طفل أيسر لها، وقيمة العنصر الجديد المراد إضافته، حيث يقوم المنهج بتكوين عقدة جديدة تحمل البيانات الجديدة بالإضافة إلى مؤشر الأب (المؤشر r) ثم جعل هذه العقدة كطفل أيسر للعقدة r عن طريق المنهج ()setLeft. سيتم إلقاء استثناء في حالة ما إن كان للعقدة طفل أيسر من قبل، كما هو موضح بالمنهج التالي:

```
public void addLeftChild(Node r, Object item) throws Exception
{
     if (r.left == null)
     {
          Node newNode = new Node(item, r);
          r.setLeft(newNode);
     }
     else
          throw new Exception (" left node is already created
     !!");
}
```

منهج إضافة طفل أيمن addRightChild

يقوم هذا المنهج بإضافة عنصر جديد كطفل أيمن لعقدة ما بنفس الطريقة التي يقوم بها المنهج السابق؛ غير أنه يستخدم المنهج ()setRight بدلا من المنهج ()setLeft، كما هو موضح بالمنهج التالي:

```
public void addRightChild(Node r, Object item) throws Exception
{
     if (r.right == null)
     {
```

```
            Node newNode = new Node(item, r);
            r.setRight(newNode);
        }
        else
            throw new Exception (" right node is already created
        !!");
}
```

مناهج عبور الشجرة Trevarsal

سنقوم بكتابة ثلاثة مناهج لعبور الشجرة كالتالي:

1- منهج العبور الوسطي للشجرة inOrderTrevarsal

نستخدم هنا الاستدعاء الذاتي للمنهج بحيث يتم زيارة العقد في الفرعي الأيسر للشجرة أولا ثم المعالجة (وهي هنا الطباعة) ثم زيارة الفرعي الأيمن للشجرة مستخدمين الاستدعاء الذاتي للمنهج، كما موضح بالمنهج التالي:

```
public void inOrderTrevarsal(Node t)
{
    if (t != null)
    {
        inOrderTrevarsal(t.left);
        System.out.print(t.data + "          ");
        inOrderTrevarsal(t.right);
    }
}
```

2- منهج العبور القبلي للشجرة preOrderTrevarsal

نستخدم هنا الاستدعاء الذاتي للمنهج أيضا بنفس الطريقة المتبعة في المنهج السابق، غير أنه تتم عملية المعالجة أولا (وهي هنا الطباعة) ثم زيارة العقد في الفرعي الأيسر للشجرة ومن ثم زيارة عقد الفرعي الأيمن للشجرة مستخدمين الاستدعاء الذاتي للمنهج أيضا وكما موضح بالمنهج التالي:

```
public void preOrderTrevarsal(Node t)
{
    if (t != null)
    {
        System.out.print(t.data + "    ");
        preOrderTrevarsal(t.left);
```

```
            preOrderTrevarsal(t.right);
        }
}
```

3- منهج العبور البعدي للشجرة postOrderTrevarsal

نستخدم هذا الاستدعاء الذاتي للمنهج أيضا بنفس الطريقة المتبعة في المنهجين السابقين، ولكن هنا يتم زيارة العقد في الفرع الأيسر للشجرة أولا، ثم زيارة العقد في الفرع الأيمن للشجرة ومن ثم تتم عملية المعالجة (وهي هنا الطباعة) مستخدمين الاستدعاء الذاتي للمنهج، كما موضح بالمنهج التالي:

```
public void postOrderTrevarsal(Node t)
{
        if (t != null)
        {
            postOrderTrevarsal(t.left);
            postOrderTrevarsal(t.right);
            System.out.print(t.data + "   ");
        }
}
```

منهج البحث عن عنصر بيانات في الشجرة search

تعتبر عملية البحث عن عنصر بيانات معين داخل عقد الشجرة من العمليات الهامة في العديد من التطبيقات. فهذا المنهج يستقبل مؤشر للعقدة المراد البحث منها وهي عادة ما تكون الجذر، بالإضافة إلى قيمة عنصر البيانات المراد البحث عنه. حيث يتم اختبار القيمة مع بيانات العقدة المرسلة أولا فإذا لم تكن مطابقة للبيانات المطلوبة يتم البحث في الفرع الأيسر للشجرة فإذا لم يتم العثور أيضا على عنصر البيانات في الفرع الأيسر للشجرة يتم الانتقال للبحث في الفرع الأيمن للشجرة. في حالة عدم وجود عنصر البيانات في الشجرة يرجع المنهج `null`، أما في حال وجود العنصر فيرجع مؤشر إلى العقدة التي تحتوي على عنصر البيانات، كما هو موضح بالمنهج التالي:

```
public Node search(Node r, Object obj)
{
        Node found = null;
        if (r != null)
        {
            if (r.data.equals(obj))
                found = r;
            else
```

```
        {
            found = search (r.left, obj);
            if (found == null)
                found = search(r.right, obj);
        }
    }
    return found;
}
```

التالي هو فصيل الـ BinaryTree ومحتويا على كل المناهج السابقة:

```
public class BinaryTree
{
    public Node root;

    public BinaryTree()
    {
        root = null;
    }

    public BinaryTree(Object item)
    {
        root = new Node(item);
    }


    public Node getRoot()
    {
        return root;
    }

    public boolean isEmpty()
    {
        return root == null;
    }

    public boolean isLeaf (Node n)
    {
        return ( (n.left == null) && (n.right == null) );
    }

    public boolean isRoot(Node n)
    {
```

```
        return (n == root);
    }

    public void addLeftChild(Node r, Object item) throws
Exception
    {
        if (r.left == null)
        {
            Node newNode = new Node(item, r);
            r.setLeft(newNode);
        }
        else
            throw new Exception (" left node is already created
        !!");
    }

    public void addRightChild(Node r, Object item) throws
Exception
    {
        if (r.right == null)
        {
            Node newNode = new Node(item, r);
            r.setRight(newNode);
        }
        else
            throw new Exception (" right node is already
        created !!");
    }

    public void inOrderTrevarsal(Node t)
    {
        if (t != null)
        {
            inOrderTrevarsal(t.left);
            System.out.print(t.data + "    ");
            inOrderTrevarsal(t.right);
        }
    }

    public void preOrderTrevarsal(Node t)
    {
        if (t != null)
```

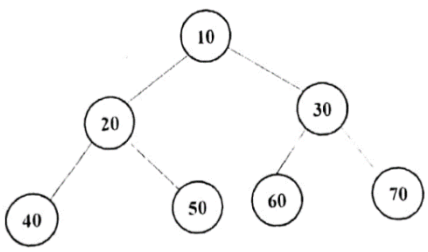
```
        {
            System.out.print(t.data + "   ");
            preOrderTrevarsal(t.left);
            preOrderTrevarsal(t.right);
        }
    }

    public void postOrderTrevarsal(Node t)
    {
        if (t != null)
        {
            postOrderTrevarsal(t.left);
            postOrderTrevarsal(t.right);
            System.out.print(t.data + "   ");
        }
    }

    public Node search(Node r, Object obj)
    {
        Node found = null;

        if (r != null)
        {
            if (r.data.equals(obj))
                found = r;
            else
            {
                found = search (r.left, obj);
                if (found == null)
                    found = search(r.right, obj);
            }
        }
        return found;
    }
}
```

البرنامج التالي يختبر هذا الفصيل ويقوم بتكوين الشجرة التالية ومن ثم طباعتها بمختلف الصيغ (inorder – preorder - postorder)



```
public class UseTree
{
    public static void main (String[] args)
    {
        try
        {
        System.out.print("\f ");

        BinaryTree tree = new BinaryTree(10);

        //System.out.println(tree.isRoot(ic));
        //System.out.println(tree.isEmpty());
        //System.out.println(tree.isleaf(it));

        Node it = tree.getRoot();
        tree.addLeftChild(it,20);
        tree.addRightChild(it, 30);

        Node it2 = it.getLeft();
        tree.addLeftChild(it2, 40);
        tree.addRightChild(it2, 50);

        Node it3 = it.getRight();
        tree.addLeftChild(it3, 60);
        tree.addRightChild(it3, 70);

        Node n = tree.search(tree.root, 40);
        if (n != null)
            System.out.println(n.getParent().data);

        //System.out.println(tree.isleft(30));

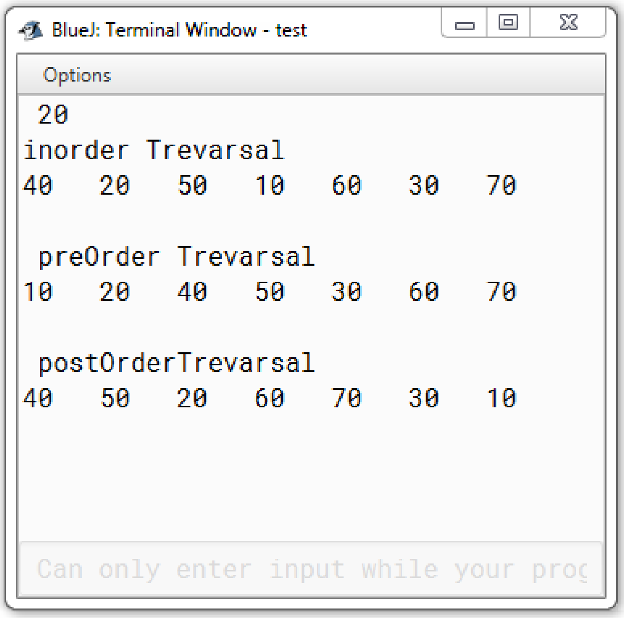
        System.out.print("inorder Trevarsal \n");
        tree.inOrderTrevarsal(tree.root);

        System.out.print("\n\n preOrder Trevarsal \n");
        tree.preOrderTrevarsal(tree.root);

        System.out.print("\n\n postOrderTrevarsal \n");
        tree.postOrderTrevarsal(tree.root);
        }
        catch (Exception e)
```

```
            {
                System.out.println(e.getMessage());
            }
        }
}
```

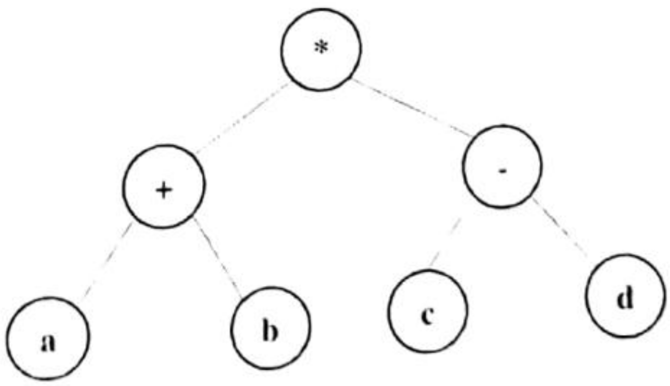
نتيجة البرنامج ستكون كالتالي:



كما يمكننا كتابة برنامج اختبار آخر لتمثيل التعبير الحسابي التالي:

(a+b) * (c-d)

والمتمثل في الشجرة التالية مع استخدام نفس فصيل الشجرة الثنائية BinaryTree السابق:



```
public class UseTree2
{
    public static void main (String[] args)
    {
        try
        {
            System.out.print("\f ");

            //                  *
            //          +               -
            //      a       b       c     d


            BinaryTree tree = new BinaryTree('*');

            Node it = tree.getRoot();
            tree.addLeftChild(it, '+');
            tree.addRightChild(it, '-');

            Node it2 = it.getLeft();
            tree.addLeftChild(it2, 'a');
            tree.addRightChild(it2, 'b');

            Node it3 = it.getRight();
            tree.addLeftChild(it3, 'c');
            tree.addRightChild(it3, 'd');

            System.out.print("inorder Trevarsal \n");
            tree.inOrderTrevarsal(tree.root);

            System.out.print("\n\n preOrder Trevarsal \n");
            tree.preOrderTrevarsal(tree.root);

            System.out.print("\n\n postOrderTrevarsal \n");
            tree.postOrderTrevarsal(tree.root);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

النتيجة ستكون كالتالي:

